View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

SCALABLE COLLECTIVE MESSAGE-PASSING ALGORITHMS

BY

PAUL D. SACK

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor William Gropp, Chair
Professor Laxmikant Kale
Professor David Padua
Professor Marc Snir

Abstract

Governments, universities, and companies expend vast resources building the top supercomputers. The processors and interconnect networks become faster, while the number of nodes grows exponentially. Problems of scale emerge, not least of which is collective performance. This thesis identifies and proposes solutions for two major scalability problems.

Our first contribution is a novel algorithm for process-partitioning and remapping for exascale systems that has far better time and space scaling than known algorithms. Our evaluations predict an improvement of up to 60x for large exascale systems and arbitrary reduction in the large temporary buffer space required for generating new communicators.

Our second contribution consists of several novel collective algorithms for Clos and torus networks. Known allgather, reduce-scatter, and composite algorithms for Clos networks suffer the worst congestion when the largest messages are exchanged, damaging performance. Known algorithms for torus networks use only one network port, regardless of how many are available. Unlike known algorithms, our algorithms have a small amount of redundant communication. Unlike known algorithms, our algorithms can be reordered so that congestion hinders small messages rather than large, and all ports can be fully used on multi-port torus networks. The redundant communication gives us this flexibility. On a 32k-node system, we deliver improvements of up to 11x for the reduce-scatter operation, when the native reduce-satter algorithm does not use special hardware, and 5.5x for the allgather operation. We show large improvements over native algorithms with as few as 16 processors.

Doctor, doctor, is there a doctor in the house?
IS THERE A DOCTOR IN THE HOUSE?

Yes, madam, I am a doctor.

Oh, doctor, have I got a daughter for you!

–Lou Jacobi

Acknowledgments

After nine short years, this is finished.

Thank you, Jill, for your patience and for your impatience, and for always being there.

Thank you, Mom and Dad, for encouraging me always and setting a good example.

Thank you, Bill, for all the open-ended chats about supercomputing and showing me how to be a researcher. That was what I needed after my erstwhile advisor left computer science in pursuit of music superstardom.

Thank you, the rest of my committe, Sanjay, David, and Marc, for helping me shape this thesis.

Thank you, Anne, for introducing me to MPI in 2001 and sending me across the ocean to learn a little more.

Thank you Brian, Joe, Jun, Karin, Luis "el bocagrande", Pablo, and Radu for all the I-ACOMA hijinx. Perhaps the simulator-support emails will cease when I am no longer `paulsack@uiuc.edu`.

Thank you, Rhonda and Mary Beth for guiding me through the finish line.

Thank you, people of Argonne National Labs who tend the most pleasantly-configured system I have ever had the pleasure to use. And finally:

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by the U.S. Department of Energy under contract DE-FG02-08ER25835 and by the National Science Foundation under grant 0837719.

Table of Contents

List of Figures

Chapter 1

Introduction

Petascale supercomputers, *i.e.*, supercomputers that can sustain over a Petaflop of performance were once unimaginable; now the top 10 supercomputers in the world have broken the petaflops barrier [2]. The scalability problems faced by the designers of these systems will be familiar to those who will build exascale supercomputers in the not-so-distant future. These designers will face an additional challenge in obtaining high-performance collective-communication operations.

A typical supercomputer near the top of today's Top 500 list would be built with multicore microprocessors with a shared cache. Several of these would be placed on one board and share memory. Several dozen boards would be placed in a cabinet. Then many cabinets would be connected. The interconnects used by the systems near the top are evenly split between proprietary 3d-torus networks and Infiniband fat trees.

Engineers design systems for efficient point-to-point communication, especially between nearby processors. Collective communication is less frequent, *e.g.*, a convergence test at the end of each iteration in an iterative algorithm. As systems become larger, collective communication performance increasingly suffers compared to local point-to-point communication. That torus networks are still attractive for many workloads illustrates the primacy of point-to-point communication performance.

This thesis explores two important areas in collective communication: process partitioning and remapping, and collective operations. With our novel algorithms, supercomputers can deliver better communication performance without any additional hardware costs or runtime or compiler support. Our algorithms can be simple additions to any MPI library.

In the next chapter, we explore how known approaches for communicator-creation, the mechanism by which processes are partitioned or remapped, do not scale, and we propose straightforward algorithms that rectify the situa-

tion. On an n-processor system, we improve the latency of communicator-creation from $O(n \lg n)$ to $O(p \lg n + \lg^2 n + n/p \lg p)$ and the memory requirement from $O(n)$ to $O(n/p)$. (p is a parameter used in our algorithm, and $p \ll n$ for optimal performance.) On a large exascale system with 128 million processor cores, we reduce the time spent in creating a new communicator sixty-fold, from 22 seconds to 370 milliseconds.

The largest supercomputer today, the K computer, has just over half a million cores [3]. The scalability problems in communicator management are not serious on supercomputers with under several to tens of millions of cores. More pressingly, current algorithms for common collective-communication operations do not deliver good performance on even small systems, such as a 512-node partition of a BlueGene/P supercomputer.

The models used in developing collective algorithms generally either ignore network topology altogether or, equivalently, assume that the delivered bisection bandwidth meets or exceeds the injection bandwidth. When practical topologies are considered, performance declines considerably for these algorithms. Known algorithms developed specifically for torus networks can scale well, but only use one of the six ports typically available in 3d torus networks.

Even on networks where bisection bandwidth meets injection bandwidth, in practice, network performance suffers at high network loads, and these networks deliver somewhat less than their theoretical bisection bandwidth [4, 5]. Moreover, full bisection bandwidth is not a given. The Roadrunner system, currently the tenth-fastest in the world, has a two-level Clos ("fat-tree") network, where the bisection bandwidth at the top level provides for just over 25% of the injection bandwidth from the processor cores into the lower level. The systems with toroidal interconnects of course have far less bisection than injection bandwidth. The plans for the Blue Waters system had a three-layer fat tree-like network and a bisection bandwidth of 85% of injection bandwidth [6].

Our solution, detailed in Chapter 3, consists of loosening the restriction on data ordering. This allows for better communication locality in Clos and torus networks, and for the use of all ports in multi-ported torus networks. We then add a small amount of redundant communication to restore the correct ordering. This speeds up allgather operations by up to 5.5x; and reduce-scatter operations by up to 11x compared to the native algorithm

on a BlueGene/P system, when the native reduce-scatter algorithm is not using the special tree network which does reductions in place. Our work can also improve the performance of the broadcast, reduce, and reduce-scatter operations.

This thesis makes two main contributions. First, we identify and fix the scaling problem in process-partitioning and remapping algorithms.

Second, we present algorithms for important collective-communication operations that significantly improve performance through redundant communication.

This work is divided into two chapters. In Chapter 2, we share our work on scalable process partitioning and remapping. In Chapter 3, we demonstrate our novel collective-communication algorithms.

Chapter 2

Scalable process partitioning and remapping algorithms

2.1 Introduction

The Message Passing Interface Forum began developing the message-passing interface (MPI) standard in early 1993. MPI defines a standard communication interface for message-passing systems that includes support for point-to-point messaging, collective operations, and communication-group management. The core of communication-group management is the *communicator*: a context in which a group of processes can exchange messages.

In 1993, the fastest supercomputer in the Top 500 was a 1024-processor Thinking Machines system that could sustain nearly 60 Gigaflops [2]. In the June 1997 list, the 7264-processor Intel ASCI Red system broke the Teraflop barrier. In early 2010, the fastest supercomputer was a Cray system with 224,000 cores at 1.8 Petaflops, and a Blue Gene system ranked fourth has nearly 300,000 cores achieves nearly 1 Petaflop. By mid-2011, the fastest supercomputer, the K computer, attained 8 Petaflops with 550,000 cores. If scaling trends continue, supercomputers with *millions* of cores will achieve Exaflop performance. Current methods for managing communicators in MPI do not perform well at these scales – in space or in time – and programmers cannot program their way around using communicators.

In this chapter, we first examine the state of the art in Section 2.2 and identify the MPI communicator-creation functions that are not inherently unscalable. In Section 2.3, we propose novel communicator-creation algorithms that do scale to million-core supercomputers. In Section 2.4, we detail our evaluation methodology and present our results. We conclude in Section 2.5.

2.2 Background

MPI programs start with one communicator, `MPI_COMM_WORLD`, that contains every process in the program. It is common to form smaller communicators containing a subset of all the processes. MPI provides two ways to do this:

- `MPI_Comm_create`: processes must enumerate all the members of the new communicator.
- `MPI_Comm_split`: processes specify a color; processes whose colors match become the members of new communicators.

`MPI_Comm_create` requires as input a table specifying membership in the new communicator for every process in the old communicator. Only one communicator is created per call (in MPI 2.1), and ranks can not be reordered. Each process in the old communicator must call this function with the same table, whether or not the process is included in the new communicator.

This algorithm scales poorly as the memory and computation costs scale linearly with the size of of the input communicator.

`MPI_Comm_split` requires only a color and a key as input. (The key is used to reorder ranks in new communicators.) It is more versatile, since it can create many communicators in one call, can reorder ranks, and does not require the programmer to build a large table to specify communicator membership. The ranks in the new communicator are assigned by sorting the keys and using the ranks in the old communicator as a tie-breaker.

We examined the implementation of `MPI_Comm_split` in two widely-used open-source MPI implementations: MPICH[7, 8] and OpenMPI[9], which is derived from LAM-MPI. Unfortunately, `MPI_Comm_split` scales equally poorly in these current MPI implementations, which operate as follows:

1. Each process builds a table containing the colors and keys of all the other processes with an allgather operation.
2. Each process scans the table for processes whose colors match.
3. Each process sorts the keys of processes whose colors match.

MPICH makes use of "recursive doubling" to build the table [10]. In the first step, each process exchanges its color and key information with the

process whose rank differs only in the last bit. In the second step, each process exchanges its color and key table (now containing two entries) with the process whose rank only differs in the second-last bit. This continues, and in the final step, each process exchanges a table containing $n/2$ entries with the process whose rank differs only in the first bit.

Open MPI uses an operation similar to recursive doubling to build the table, known as Bruck's algorithm, which works whether or not the number of processes is a power of two. (MPICH uses this algorithm only when the number of processes is not a power of two.) We explain it in greater detail in Section 3.2.2. We assume the use of recursive doubling in the rest of this work, but it makes little difference in the analysis which is used.

In both implementations, after the tables are built, the tables are scanned for entries with matching colors, and then those entries are sorted by key.

If there are n processes in the input communicator, this algorithm incurs a memory and communication cost of $\Theta(n)$, and a computation cost dominated by the sorting time: $O(n \lg n)$. Only the entries in the table whose colors match are sorted, so for smaller output communicators, the computation cost can be much less.

As we will show, the sorting phase of `MPI_Comm_split` consumes a significant amount of time at larger scales. Further, the table requires vast amounts of memory. Every entry has at minimum three fields: hardware address or rank in the source communicator; color; and key. Using 32-bit integers for each field, this requires 192 Megabytes per process per communicator in a 16-million node system, which is clearly unreasonable. This is especially problematic because per-core performance is growing far more slowly than the number of cores in modern microprocessors [11], and we can only expect memory per-core to grow as quickly as performance per-core for weak scaling.

2.3 Scalable communicators

While the memory problem is more concerning, we first present one facet of our solution to the poor performance scaling of `MPI_Comm_split`.

2.3.1 Better performance through parallel sorting

As mentioned, much of the time in `MPI_Comm_split` is spent in sorting the color and key table. In state-of-the-art algorithms, every process sorts the entire table. Our proposal is to sort this table in parallel. The simplest and least-effective way to do this is to have each process sort its table at each step in the recursive doubling operation before exchanging tables with another process. In effect, at each stage, each process merges the sorted table it already has with the sorted table it receives from its partner process for that stage. This turns the $O(n \lg n)$ sort problem into $\lg n$ merge operations of size $2, 4, 8, \ldots, n/4, n/2$, an $O(n)$ problem.

This gives us an $O(\lg n)$ speedup. In the final step, all n processes are merging identical tables. In the second-to-last step, two groups of $n/2$ processes merge identical tables. To do away with this redundant computation, we adapt a parallel sorting algorithm to this problem.

Cheng *et al* present one parallel sorting algorithm [12]. Their algorithm begins with an unsorted n-entry table distributed among p processes. At the conclusion of the algorithm, the table is sorted, and the entries in the the ith process's table have global rank $[(i-1)\frac{n}{p}, i\frac{n}{p})$.

In brief, their algorithm works as follows:

1. Each process sorts its local table.

2. The processes cooperate to find the exact splitters between each process's portion of the final sorted table using the exact-splitter algorithm from [13].

3. The processes forward entries from their input subtables to the correct destination process's subtable using the splitters.

4. Each process merges the p pre-sorted partitions of their subtable.

In step 1, each process takes $O(\frac{n}{p} \lg \frac{n}{p})$ time to sort its subtable. However, we use recursive doubling and merging to generate the inputs, thus we need not sort the subtable. Recursive doubling up to the beginning of the parallel sort, where each process has a table with $\frac{n}{p}$ elements, takes $O(\frac{n}{p})$ time.

In step 2, exact splitters are found after $\lg n$ rounds in which the distance between the estimate of the splitter and the correct splitter exponentially

decays. Each round has an execution-time cost of $O(p + \lg \frac{n}{p})$, for a total time $O(p \lg n + \lg^2 n)$.

In step 3, each process exchanges at most $\frac{n}{p}$ entries with other processes. (For random input keys and ranks, the number of entries is expected to be $\frac{n}{p} \times \frac{p-1}{p}$.) Thus, the time complexity for the exchange is $O(\frac{n}{p})$.

In step 4, each process recursively merges the p partitions of its subtable in $\lg p$ stages, merging p partitions into $p/2$ partitions in the first stage, then $p/2$ partitions into $p/4$ partitions, and so on, for a cost of $O(\frac{n}{p} \lg p)$.

At the conclusion of the parallel sort, the p processes use recursive doubling to build full copies of the table.

The total time spent in the parallel sort is $O(\frac{n}{p}) + O(p \lg n + \lg^2 n) + O(\frac{n}{p}) + O(\frac{n}{p} \lg p) = O(p \lg n + \lg^2 n + \frac{n}{p} \lg p)$. There is no closed-form expression for p in terms of n that minimizes time. If $O(1) < p < O(\frac{n}{\lg n})$, then, asymptotically, the parallel sort will take less time than the $\lg p$ stages of merging in the conventional algorithm with merging.

The time spent in the final recursive-doubling stage to build full copies of the table is $O(n)$.

The amount of communication is more than that in a conventional recursive-doubling implementation due to the communication incurred in finding the splitters and exchanging table entries between sorting processes. Even so, we observe performance improvements with a modest number of sorting processes.

2.3.2 Less memory usage with distributed tables

For an application to scale to millions of processes, each process must only exchange point-to-point messages with a small number of unique processes (*e.g.*, $\lg n$ processes for the recursive-doubling allgather algorithm). Thus, most of the entries in a large communicator table go unused.

We propose that groups of P' processes share a full table, where each process stores n/P' rows of the full table. When a process needs the address of a process not in its portion of the table, it queries the process in the group which has the address in its table. Each process will also maintain a small cache with the results of these remote lookups. This can be implemented with efficient one-sided communication.

The number of processes in each group can be the same, less than, or greater than that used in the parallel sort. In large exascale systems with scarce memory, the memory usage of the table can be arbitrarily tightened by increasing the size of this group.

2.3.3 Related work

When we published our results in [14], it was the first work to address process partitioning and remapping at large scales. Later, Moody *et al* [15] and Siebert and Wolf [16] looked at scalable `MPI_Comm_split` algorithms.

In Moody *et al*'s approach, communicators are stored as doubly-linked lists. Each process in a communicator knows only the addresses of its two adjacent rank neighbors. They present a method for implementing binary-tree collective algorithms in $\lg n$ steps, provided that the binary-tree operation is *distance-doubling*, *i.e.*, the distance between communicating ranks doubles in each step. This precludes the use of better collective algorithms, as we explain in Chapter 3. The linked-list structure also precludes efficient point-to-point message-passing, according to the authors.

They present algorithms for the general `MPI_Comm_split` problem, the case when only the ranks are reordered and the communicator is not split, and the case when the communicator is split but ranks are not reordered. For the general problem and the reordering-only problem, their algorithms include a radix sort or a bitonic sort. It is not clear how they are implemented without efficient point-to-point messaging.

They compare their algorithms for sorting-only (and not splitting) with our merging-in-place algorithm and our first parallel-sorting algorithm, with replicated, not distributed tables. We will compare our results with theirs in the next section.

Siebert and Wolf present a novel parallel sorting algorithm that can be used to sort the color-key pairs in `MPI_Comm_split`. Unlike the Cheng algorithm we adapt, where each sorting process is responsible for sorting many color-key pairs, in their algorithm, each process is only responsible for one color-key pair.

Their algorithm has a divide-and-conquer approach. First, all the processes participate in finding a good splitter. Using global-sum reductions and scan

operations, each process can determine how many color-key pairs fall on each side of the splitter. The problem is then split in two, and color-key pairs are exchanged between processes so that all color-key pairs less than the splitter are in one group and all color-key pairs greater than the splitter are in the second group. This iterates for $O(\lg n)$ stages with high probability for a total runtime of $O(\lg^2 n)$. Essentially, this is an extremely-parallel quick sort. They develop specialized reduction and scan collective operations that operate on ranges of processes to avoid creating new communicators.

The result is that the process with rank i in the input communicator has the globally-sorted color-key pair of rank i.

Their algorithm requires far less communication and only a small amount of temporary buffer space. They do not explain how to build a communicator using the sorted output. In [14], we proposed a tree-like structure to store the communicator, where each process has the rank-address mapping of itself, its children, and its parent, along with a small rank-address cache. Rank-address lookups consist of, at worst, a $2 \lg n$-hop traversal on the tree. This tree could be built using the scattered result from their algorithm quite easily.

We later decided that distributing the full rank-address table among groups of p processes would be a simpler, more-efficient approach with acceptable memory usage. This could be implemented after their algorithm has completed using a recursive-doubling allgather operation that stops several stages early. This would eliminate the space advantage of their algorithm and the asymptotic runtime advantage, however, even at the exascale, constant factors are very important, and a more efficient sort than the Cheng sort we use is still of great benefit.

There are many other parallel sorting algorithms we could use instead. Many have a similar structure to ours but use inexact splitters, *e.g.*, [17]. Solomonik and Kale [18] loosely follow the structure of the Cheng algorithm but use inexact splitters and overlap all four stages. Processes sort their subtables and find splitters at the same time. Some splitters will be found before others; for those, the data forwarding can begin early. Each process will receive data from some processes before others; the merging can begin before all the data is available. This overlaps communication and computation and is likely to be far more efficient than the algorithm we use.

2.4 Evaluation

There are no systems yet with millions of processors, so we investigated the possibility of simulation. All but the parallel sort with distributed tables algorithm entail each of n processes receiving an n-entry table to sort. This requires simulating $n^2/flit\text{-}size$ flits.

BigNetSim can simulate up to 2.5 million network events/second using 128 processors [19]. Every hop in the network counts as an event. Thus, one experimental data point would require simulating billions of events and take months.

Instead, we use an indirect approach. We fully implement the parallel-sort phase of the algorithm using two through 64 processes. The recursive-doubling exchanges, and the sorting and merging in the conventional algorithm, we simulate using only two processes. These two processes exchange, merge, or sort the correct amount of sorted or unsorted data with each other at each iteration in each phase of `MPI_Comm_split`. In the correct algorithm, each process would exchange data with another unique process at each level in the recursive-doubling algorithm.

We do not capture the effect of network contention. Network contention would have the greatest effect during the final stages of the conventional algorithm, when every process exchanges messages on the order of several to tens of Megabytes. Not modeling network contention favors the conventional algorithm in our evaluation. However, in Chapter 3, we present topology-aware allgather algorithms that avoid congestion and deliver high performance.

In our evaluation, we generate one new communicator containing all the processes in the input communicator. The keys are randomly-generated. Later, we discuss how the results are likely to change for smaller communicators.

The experiments are run on Blueprint, a system of 120 POWER5+ nodes, each with 16 1.9 GHz Power 5 cores. The nodes are connected with Federation switches. In all of our tests, we use only one core per node to better simulate a million-core system. We use merge and sort functions from the C++ STL, and compile with IBM XLC at optimization level `-O4`.

Each configuration was run 5 times and the mean is shown. The scale of the standard error was too small to appear on our graphs.

Speedup data for our two parallel-sort `MPI_Comm_split` variations are re-

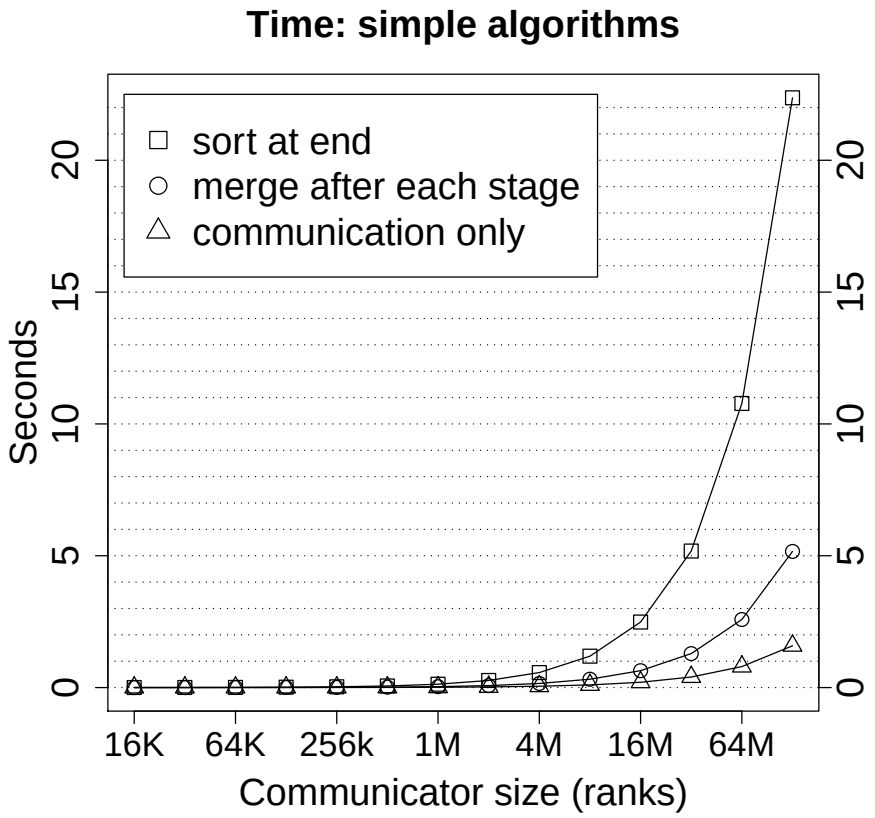

Figure 2.1: The total execution time of the conventional algorithm, the conventional algorithm with merging after each step, and the communication time for both.

ported relative to to the conventional `MPI_Comm_split` variation with merging after each step.

In Figure 2.1, we see the latency of the conventional `MPI_Comm_split` algorithm implemented in MPICH and Open MPI, along with the time spent in communication for both. The time for one `MPI_Comm_split` call is reduced by a factor of four simply by replacing the $O(n \lg n)$ sort at the end with an $O(n)$ (total) merge after each step.

Figure 2.2 shows the speedup if we sort the color and key table in parallel and then continue using recursive doubling to create a full copy of the new tree. At best, we get a 2.7x advantage over serial sort.

Figure 2.3 shows the speedup if we sort in parallel and build the communicator as a distributed tree. We get performance benefits of up to 14x using 64 sorting processors. Compared to the sort-at-the-end MPICH and

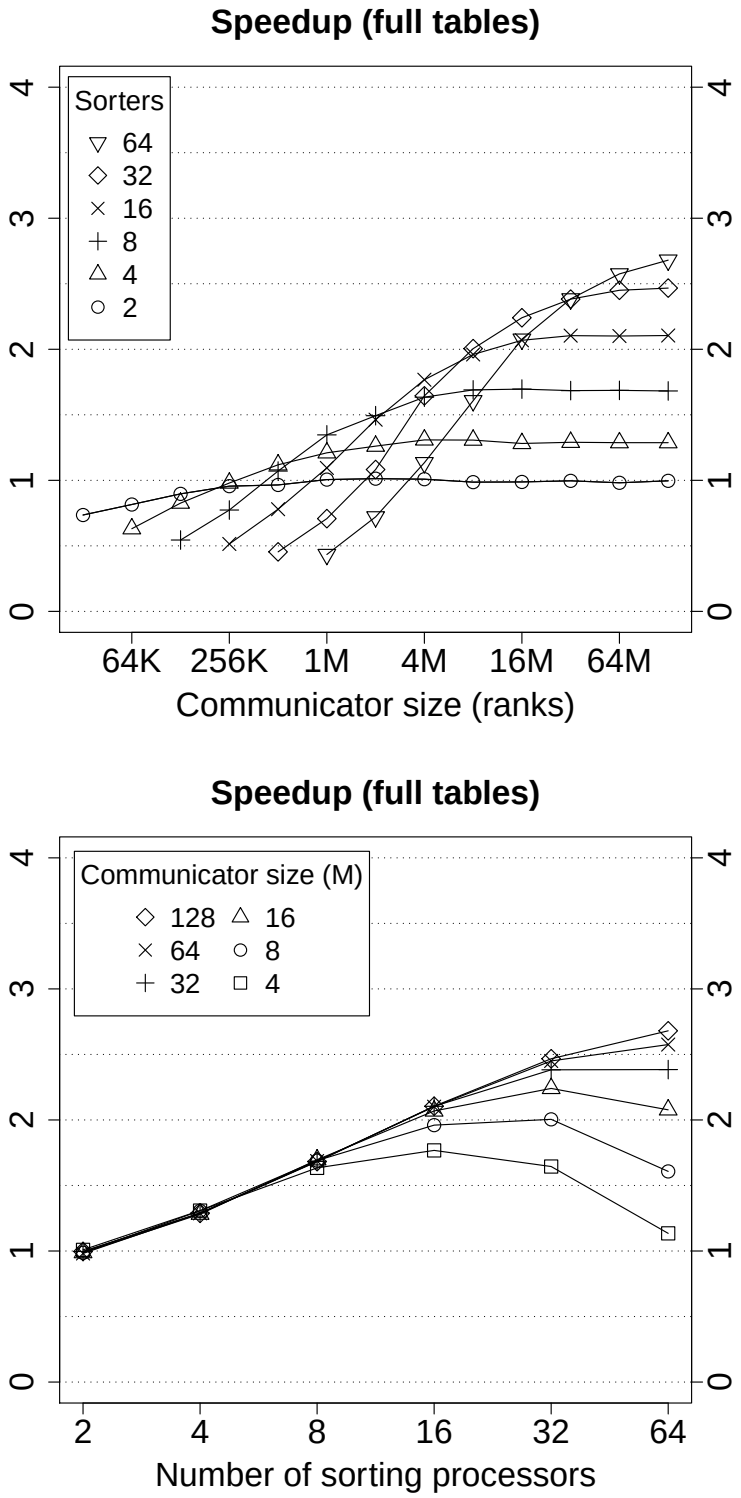

Figure 2.2: The speedup of using a parallel-sort with conventional communicator tables over merging in-place.

Open MPI implementation, our algorithm is 60x faster. The parallel sort with distributed communicators algorithm significantly reduces the amount of communication and network contention, so the speedups on real systems may be better.

More important than a single performance number is the analysis of where the time goes. Table 2.1 gives a breakdown of the time spent in each operation for the parallel-sort algorithm with distributed tables.

The breakdown matches expectations perfectly. The *collect* rows show the time spent in the recursive-doubling stage before the parallel sort. As expected, it is proportional to the size of each sorting process's subtable: *i.e.*, the number of processors in the `MPI_Comm_split` call divided by the number of sorting processes.

The *splitters* rows show the time spent in calculating the median-of-medians. The number of iterations in the exact-splitter algorithm increases as the logarithm of the size of the problem per process. (The distance between the chosen splitter and the exact splitter exponentially decreases as a function of the iteration count.) The time spent per iteration is dominated by a term linear with the number of sorting processes. The time spent in this stage is insignificant until we scale to 32 or more sorting processes.

The *exchange* rows show the time spent exchanging elements between sorting processes once the exact splitters are found. With p sorting processes, on average, we expect $p - 1$ out of every p entries on each process will be exchanged. Thus, this term grows as the size of each sorting process's subtable.

The *merge* rows show the time spent merging the p partitions of each process's subtable together. In the first iteration, $\frac{p}{2}$ pairs of $\frac{n}{p^2}$-entry subtables are merged, in the second iteration, $\frac{p}{4}$ pairs of $\frac{2n}{p^2}$-entry subtables are merged, and so on, for a total cost of $O(\frac{n}{p} \lg p)$ in $\lg p$ iterations. We observe that the times for 2 or 4 sorting processes are the same, since the $\lg p$ term doubles as the $\frac{n}{p}$ term halves.

2.4.1 Comparison with related work

In Section 2.3, we describe Moody *et al*'s work and Siebert and Wolf's work on scalable `MPI_Comm_split` algorithms. Moody *et al*'s best algorithm is

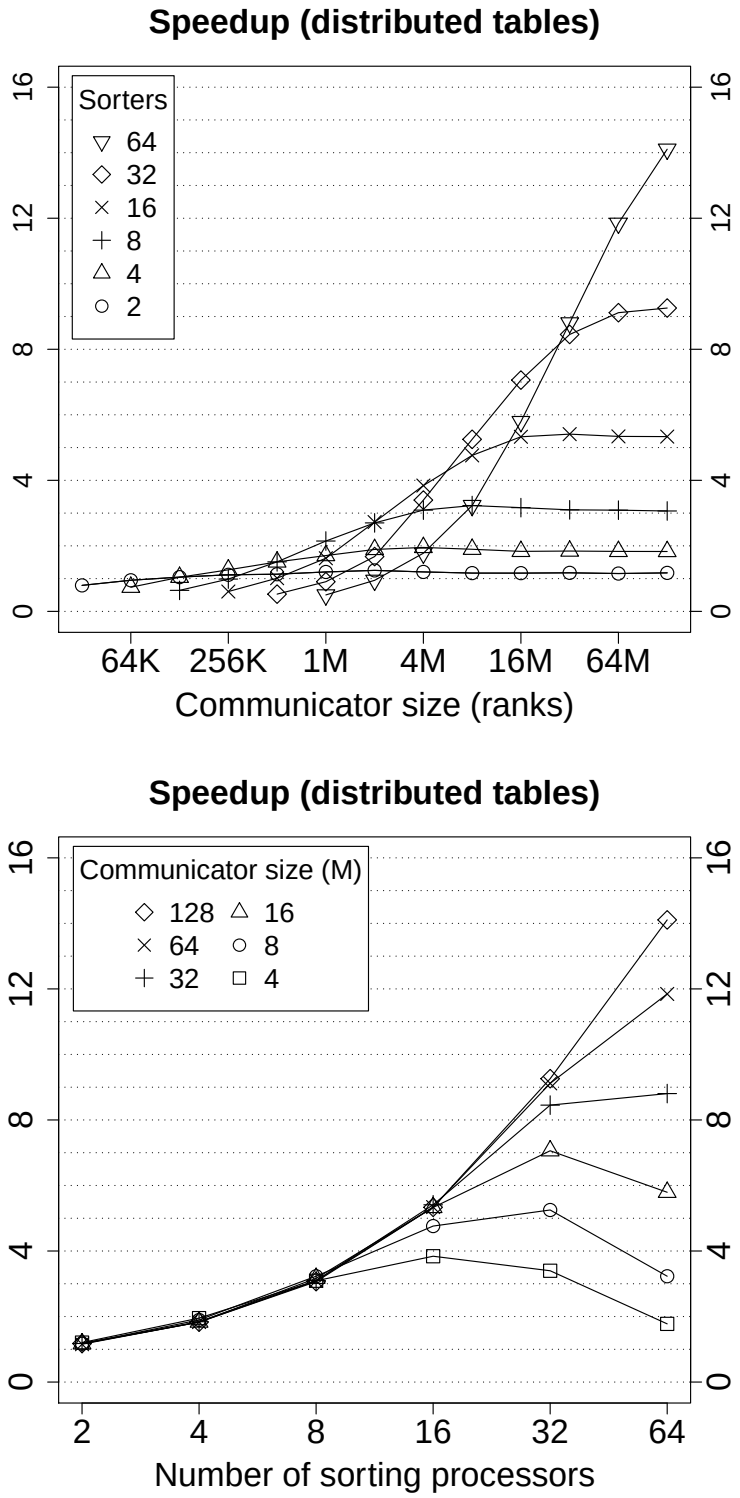

Figure 2.3: The speedup of using a parallel-sort with distributed-tree communicators over merging in-place.

Communicator size	Operation	Sorting processors					
		2	4	8	16	32	64
8M	collect	0.16	0.08	0.03	0.02	0.01	0.00
	splitters	0.00	0.00	0.01	0.01	0.03	0.08
	exchange	0.05	0.03	0.02	0.01	0.01	0.01
	merge	0.06	0.06	0.04	0.03	0.02	0.01
	total	0.27	0.17	0.10	0.07	0.06	0.10
16M	collect	0.32	0.16	0.08	0.03	0.02	0.01
	splitters	0.00	0.00	0.01	0.01	0.03	0.08
	exchange	0.10	0.06	0.03	0.02	0.01	0.01
	merge	0.13	0.13	0.09	0.05	0.03	0.02
	total	0.55	0.35	0.20	0.12	0.09	0.11
32M	collect	0.64	0.32	0.16	0.08	0.03	0.02
	splitters	0.00	0.00	0.01	0.02	0.03	0.08
	exchange	0.20	0.12	0.07	0.04	0.02	0.01
	merge	0.25	0.26	0.19	0.11	0.07	0.04
	total	1.09	0.70	0.42	0.24	0.15	0.15
64M	collect	1.29	0.64	0.32	0.16	0.08	0.03
	splitters	0.00	0.00	0.01	0.02	0.03	0.08
	exchange	0.44	0.24	0.13	0.07	0.04	0.02
	merge	0.51	0.52	0.38	0.24	0.14	0.08
	total	2.24	1.41	0.84	0.48	0.28	0.22
128M	collect	2.58	1.29	0.64	0.32	0.16	0.08
	splitters	0.00	0.00	0.01	0.02	0.03	0.08
	exchange	0.80	0.48	0.27	0.14	0.07	0.04
	merge	1.01	1.05	0.77	0.49	0.29	0.17
	total	4.39	2.82	1.69	0.97	0.56	0.37

Table 2.1: Breakdown of time (seconds) in `MPI_Comm_split` with parallel sorting and distributed tree communicators.

4.4 times faster than their implementation of our parallel sort, conventional-tables algorithm. Our distributed-tables algorithm is over 5 times faster than our conventional tables algorithm. Moreover, point-to-point messaging is supported with our distributed communicator structure, whereas their linked-list structure prevents efficient point-to-point messaging and restricts algorithms for collective operations.

They implement our algorithm with 128, 512, 2048, or 8192 sorting processes. These four variations take a different amount of time building communicators with under 128 processes–it is not explained how this is possible. Moreover, they show communicators on a Blue Gene/P built in 20 microseconds, when the ping-pong latency is 3 microseconds.

Siebert and Wolf evaluate their new sorting algorithm against several alternatives on a Blue Gene/P using up to 294,912 processes on 73,728 quad-core nodes. They extrapolate their performance to 128 million processes to compare against our results. On the 128-million process problem, their algorithm is predicted to sort the color-key pairs in 12.7 ms, whereas our time at that scale was 370 ms. Their algorithm requires only a few dozen bytes of memory, whereas ours requires a few megabytes. Recall that Siebert and Wolf's sort leaves one globally-sorted color-key pair on each process. They do not fully explain how a communicator structure can be built from this. In Section 2.3, we present a couple of possibilities for how this might be done.

In all fairness, it is not especially meaningful to compare performance numbers between the two new approaches given the data available; they implement their `MPI_Comm_split` algorithms on a Blue Gene/P and build communicators with up to 64k processes or 294k processes, whereas we evaluate exascale supercomputers with 8 million or more processes.

2.4.2 Small communicators

We experimented with the case where we take one very large input communicator and create a very large output communicator of the same size but a different rank order. The conventional algorithm collects the entire color and rank table and then only sorts the entries whose color matches that of the desired new communicator. Thus, when the size of the output communicators is much smaller, the performance for the conventional algorithm will

not be nearly as bad as that shown here.

However, observe that ignoring the sort cost, the conventional algorithm still spends nearly 2 seconds just in communication time for a 128 million-process input communicator. Our parallel algorithm with 128 sorting processes takes 0.37 seconds, including the sort. Moreover, our algorithm does not need 128 million entries worth of temporary storage in each process.

2.5 Conclusion

Existing algorithms for creating MPI communicators do not scale to exascale supercomputers containing millions of cores. They use $O(n)$ memory and take up to $O(n \lg n)$ time in an n-process application, whereas per-core performance and memory are expected to increase sub-linearly in the future.

Our work proposes three techniques that solve the time and space scalability problem: merging after each step in the allgather phase, sorting in parallel, and using distributed rather than replicated tables.

These techniques together reduce the time complexity to $O(p \lg n + \lg^2 n + \frac{n}{p} \lg p)$, where we have groups of p sorting processes, and reduce the memory footprint p-fold. In our experiments, this reduces the cost of `MPI_Comm_split` from over 22 seconds for the largest problem to just 0.37 seconds, a 60x reduction.

Chapter 3

Scalable collective communication

Point-to-point operations in MPI allow pairs of individual processes to exchange messages. The MPI standard also defines a handful of collective operations which involve all the processes in a communicator. Known algorithms for collective communication in MPI have been designed to minimize data movement, across the network and in memory, with few exceptions. The output of any collective operation must be in a certain order. These constraints curtail the options for collective algorithms. In this work, we show that a small amount of redundant communication can have an enormous effect on performance by reducing congestion and allowing for multi-port algorithms on networks that support multi-port communication.

There are dozens of algorithms for these operations. Some are topology-aware and some are not. Some work better for larger messages on smaller systems and some work better for smaller messages on larger systems. All but one known algorithm for large collective operations are minimal with respect to the total number of bytes exchanged between all the processes. *E.g.*, in every known broadcast algorithm, if the root process broadcasts a message of n bytes, every process besides the root process receives exactly n bytes and no more.

The only known non-minimal collective algorithm is that often used in the short allreduce operation and is described later.

Many supercomputers with Clos ("fat-tree") networks have less bisection bandwidth than injection bandwidth. The Roadrunner system, currently at number 7 on the Top 500 list [2], has a two-level tree, where the top level provides for just over 25% of the injection bandwidth from the processor cores into the lower level. All supercomputers with torus networks have less bisection bandwidth than injection bandwidth. In other words, under certain bisect communication patterns, congestion will limit performance.

Unfortunately, many widely-used collective communication algorithms make

use of bisect patterns that cause the worst possible congestion. Also, multi-ported switches on torus networks allow for nodes to exchange messages on multiple links simultaneously, but collective algorithms do not take advantage of this.

To avoid these problems, some supercomputer vendors write their own versions of collective algorithms that are topology-aware or use special hardware features. *E.g.*, IBM's Blue Gene series of supercomputers have a 3d torus and a tree network, and collective algorithms use whichever network is best [20, 21]. The network switches also have special features, including a provision for messages containing a "deposit bit" which are then broadcast to every node along a row or column in the torus. The tree network is only available on collective operations performed on an entire partition. Some features, like the deposit bit, are only available on contiguous subpartitions. The MPI library selects the best algorithm for each operation based on the form of the communicator, be it a full partition; a contiguous subpartition; or scattered, non-contiguous processes. The Blue Gene system provides a simple interface for programs to query topology information.

On the other hand, other supercomputer vendors ship job schedulers that ignore topology altogether, making topology-aware programming difficult [22]. This exacerbates the problem, because congestion can arise from different jobs competing for the same network links in addition to the use of poorly-performing collective algorithms.

Our work diverges from known work by adding a small amount of redundant communication to collective algorithms. This gives us more flexibility in how the algorithms proceed. The benefit is reduced network congestion, the ability to fully use multi-ported networks, and, ultimately, much better performance.

The essence of our approach is that we temporarily relax the requirement that the output of an operation be in a particular order as specified in the MPI standard. Instead, we require only that the output be in the same order on every process. We then add a small extra stage of communication either before or after the operation that restores the correct ordering. On a P-node system, this increases the amount of communication by a factor of only $1/P$ to $2/P$. On the other hand, it eliminates or sharply reduces congestion and allows the use of multi-port algorithms.

Specifically, for all networks, we reorder the communication stages in the

recursive-doubling allgather and recursive-halving reduce-scatter algorithms, and, for mesh or torus networks, we execute three or six three-dimensional bucket operations in parallel. This produces the correct result in the wrong order; one extra stage of communication restores the correct order.

These non-minimal algorithms deliver better performance on any network that can suffer from congestion or allows multi-ported communication. In tests on an IBM Blue Gene/P, our best algorithm delivers up to 6x better performance for allgather, and up to 11x better performance for reduce-scatter over the native algorithm. For larger problems, broadcast, reduce, and allreduce are usually decomposed into a scatter and allgather, reduce-scatter and gather, or reduce-scatter and allgather, respectively. Our methods improve the performance of these operations as well. We have not tested our algorithms for the Clos network, but, according to our model, bandwidth should be improved from just over the bisection bandwidth with known algorithms to nearly the full injection bandwidth with our algorithm.

In Section 3.1, we present the network model used in our analysis. In Section 3.2, we discuss related work and known algorithms. In Section 3.3, we analyze the performance of known allgather and reduce-scatter algorithms on Clos and torus networks. In Section 3.4, we introduce our novel non-minimal algorithms. In Section 3.5, we evaluate the performance of our algorithms on a 32k-node supercomputer, and in Section 3.6, we discuss other potential non-minimal algorithms and conclude.

3.1 Model

Our work is based on a common three-parameter model commonly used to analyze collective-communication algorithms. The three terms are α: the startup cost per message, which incorporates the time between when a process initiates sending a message and the first byte is available to the receiving process; β, the bandwidth cost, which is proportional to the size of each message and represents the bandwidth constraints of each link in the system, and γ which is the cost to apply the reduction operation to two elements. This model is useful to predict performance on systems with wormhole routing, where the distance between communicating pairs of processes has little effect on the latency of exchanging a message.

We ignore the γ term, as our work focuses on network performance and has no bearing on the computation time spent in reduction operations.

We use two performance models based on this three-parameter model. Both consider algorithms divided into a series of stages, where each process cannot begin the next stage until all the processes have completed the previous stage.

We analyze *single-port* networks, in which each process can send and receive one message at a time, and *multi-port* networks, where each process can send and receive as many messages as each node has links. We assume there is only one MPI process on each node. Multiple processes on a node are best handled using a hierarchical MPI library, such as [25].

No-congestion model: The first model ignores topology and congestion. β represents the bandwidth with which each processor is connected to the network. Each process can only send and receive one message in each stage in this model. The α term for an algorithm is simply the number of stages, since each process can exchange up to one message per stage. The β term is found by summing the maximum message size per stage over all the stages. This model can also be seen as one in which each processor is connected by one link to a full crossbar interconnect.

The equations derived using this model will be labeled as plain C. This is the model that was used in developing algorithms for MPICH and other work that ignores congestion and topology.

The tree broadcast algorithm, which is often used for small messages, is simple to analyze with this model. This algorithm consists of $\lg P$ stages, where there are P processes in the communicator. In the first stage, the root process sends the input message of size n to another process. In each subsequent stage, every process that has a copy of the message sends the message to a process that does not. Thus, the number of processes with a copy of the message doubles in every stage. The cost is then simply $C = (\alpha + n\beta) \lg P$.

Congestion-aware model: The second model incorporates topology and congestion. δ replaces β and represents the link speed. We add a penalty factor in the δ term to represent congestion. Topology and congestion-aware equations we will label as C_{Clos} or C_{torus}. In this model, processors on torus networks can send and receive messages on every link simultaneously.

Each processor can send and receive zero, one, or more messages per link

per stage. There *can* be dependencies between two or more messages exchanged within the same stage. A stage concludes when there is global synchronization, such as a barrier. We assume messages are routed along the shortest path or paths between source and destination.

The α term is computed by summing the maximum number of messages exchanged by any one processor in each stage over all the stages; we do not allow for overlap of the startup time with communication time when multiple messages are exchanged on different ports out of sync.

Deriving the δ term is more complex. We build a directed graph in which each edge represents a message; edges are connected when there is a direct dependency between messages. We assign a time to each edge representing the duration of the message transmission and a time to each vertex representing the completion time of the predecessor messages and the start time of the successor messages.

We start with the messages that are not dependent upon other messages. We inspect which links each message uses. For messages whose transmission does not compete for links with any other messages, we assign $T = m\delta$ to the corresponding edge, where m is the message size. For messages that do suffer from congestion, we assign $T = cm\delta$ to the edges, where c is the congestion penalty and is equal to the maximum number of messages competing for any link in the paths the competing messages take. Messages suffer from congestion if they begin at the same time and their paths share any links.

The time assigned to each successor vertex is equal to the time of the predecessor vertex plus the time assigned to each edge.

We then inspect the dependent messages which can be sent after the first round of messages have finished and assign times to each edge and vertex the same way. We iterate until no messages are left.

The δ term is then equal to the maximum end time for all the messages in the stage.

We require that messages competing for a link have the same size and start and finish transmission at the same time. We also require that messages directed towards the same vertex have the same completion time. Relaxing these restrictions would require a more detailed model and is unnecessary for analyzing the algorithms we investigate.

This model is unnecessarily complex for most of our analysis, where we could use our simple model and apply a congestion charge to each stage and

a simple change for multi-port algorithms: dividing the bandwidth term by the number of ports used. This would lead us to the same result for every algorithm but one: the 6-way bucket algorithm presented in Section 3.4.2. The main algorithm consists of three stages, and, on non-cubic partitions, the size and number of messages per link are different, and the messages on different links are independent in each stage. *E.g.*, in the second stage on an 8x16x32-node partition, 15 $8n$-byte messages will be transferred on two links, 31 $16n$-byte messages will be transferred on two links, and 7 $32n$-byte messages will be transferred on two links, with no dependencies between messages transferred on different links.

On a Clos network, each process has one link, so $\delta = \beta$, and $C_{Clos} = C$ for algorithms that do not suffer from congestion.

On a 3d torus network, each process has six links, but can only use one at a time in the simple model. Thus, $\delta = \beta$ and $C_{torus} = C$ for single-ported algorithms that do not suffer from congestion.

C_{torus} represents the cost of an algorithm on a 3d torus network. We also use $C_{2d-mesh}$ when appropriate.

Other models: Our congestion model similar to Leiserson and Maggs's distributed random-access machine (DRAM) model, which added a congestion and topology model to the common parallel random-access machine (PRAM) model [23]. The DRAM model assigns the cost of an algorithm by inspecting the amount of data transferred over each cut of a network divided by the speed of the links crossing the cut. Dependencies between messages are not modeled.

Wanker and Akerkar present a survey of many parallel computing models and the performance of algorithms on them [24]. This includes multi-dimensional mesh and torus networks and reconfigurable networks, where the links between processors can be changed dynamically. Their survey covers wormhole routing-based models like ours, where the machine size has no bearing on message latency and a model where the latency is a function of the base-2 logarithm of the number of processors in the machine.

In this work, topology and congestion are of primary concern. The δ term includes a congestion penalty for the maximum number of messages competing for the same direction on one link. We analyze two classes of networks, the 3d torus and the Clos network (colloquially known as the "fat tree"). It is simple to extend our analysis to handle a torus with any number

of dimensions, which can be equally-sized or not.

Clos model: For the Clos network, we consider networks parameterized by R, the radix of the switches, and μ, where μ is the ratio of the minimum bisection bandwidth to injection bandwidth and is constrained by: $0 < \mu \leq 1$. In an optimal Clos network, the number of up links and down links for any non-root, non-terminal switch will each be $R/2$. The root switches will each have R down links. Each bottom-level switch will be connected to T nodes and $R - T$ level-two switches, where $\frac{R-T}{T} \geq \mu$. This is the same idealized model of a Clos network as is used in [4] to minimize the number of switches on a system of a given size subject to constraints on μ and R. Switches with a radix of 128 or 256 will likely be used in supercomputers in the near future [26].

Our model for Clos networks underestimates the congestion seen in real-world Clos networks. Hoefler studied the bandwidth delivered by Infiniband Clos networks, which use static routing tables in the switches [27]. The delivered bandwidth was found to be 55-60% of that which could be delivered with ideal routing, due to hot spots. Zahavi inspected the communication patterns of all common collective algorithms on a Clos network and presented an algorithm for generating routing tables that prevent congestion in Clos networks where the bisection bandwidth matches the injection bandwidth [28]. This accelerates collective algorithms by 40%.

In our analysis, we assume that Clos networks have perfect oracular routing. We thus believe that our algorithms will improve the performance of collective algorithms on real Clos networks more than our model predicts.

3.2 Known algorithms and related work

In this work, the following operations are considered: broadcast, where one process broadcasts the same message to all the other processes; scatter, where one process sends a unique message to each process; gather, where every process sends a message to one process; allgather, where every process broadcasts a message to every other process; reduce, where a reduction operation (*e.g.*, maximum, sum, product, etc.) is applied to data from every process and the result is stored on a single process; allreduce, which is like reduce except that the result is broadcast to every process; and reduce-scatter, where the result

of the reduction is scattered among the processes.

Much thought has gone into developing fast collective-communication algorithms. Some work concerns algorithms optimized for specific network topologies; other work ignores network topologies.

3.2.1 Topology-aware algorithms

Mesh networks: In [29], a clever broadcast algorithm is developed for two-dimensional mesh topologies, which achieves $\lg P$ scaling; before, it had been thought that such topologies could achieve $\sqrt{P}$ scaling at best. Their algorithm is based on the common binary-tree algorithm. For a $p \times p$ network, in step i, each node that has a copy of the message sends it to a node that does not at a distance of $p/2^i$ hops away. This avoids the congestion inherent in running generic binary-tree algorithms on meshes. With this approach, under our two-parameter model, the cost is $C_{2d-mesh} = (\alpha + n\delta) \lg P$, where n is the size of the broadcast message.

In [30], several allreduce algorithms are analyzed and implemented on a two-dimensional mesh. These include binomial tree-based algorithms, recursive halving algorithms, a two-phase bucket-based algorithm, and several hybrids.

The simplest algorithm considered is the *fan-in, fan-out* algorithm. This is essentially a binary-tree reduce to one process followed by a binary-tree broadcast to the rest. The cost using our model is $C_{2d-mesh} = 2(\alpha + n\delta) \lg P$. This algorithm does not suffer from congestion if P is a power of two.

The next algorithm they consider is the *bidirectional exchange* algorithm. It consists of $\lg P$ stages, in which every process exchanges its full vector with another process in each stage and combines them using the reduction operator. The processes exchanging data in each step are chosen so that, at the end of $\lg P$ stages, each process has the correct output result. There is much redundant computation and communication in this algorithm, but it has fewer stages than the fan-in/fan-out algorithm and may perform better for small messages. There will be congestion in this algorithm; congestion can be minimized with clever selection of partners in each stage. This is the only known algorithm we found with redundant communication.

The cost is then $C_{2d-mesh} = \lg P\alpha + \sqrt{P}n\delta$ after accounting for congestion.

The authors then analyze the *recursive-halving* allreduce algorithm. It is composed of a recursive-halving reduce-scatter operation followed by a recursive-doubling allgather operation. The reduce-scatter and allgather operations are each composed of $\lg P$ stages. In the first stage of the reduce-scatter operation, each process exchanges and combines half of the input vector with another process. Specifically, each process sends data from one half of the vector and receives data corresponding to the other half of the vector, which is combined with the half of the vector that was not sent. In each subsequent stage, the size of the data exchanged halves. The allgather operation is the reverse. In the first stage, n/P elements are exchanged, in the second $2n/P$, and so on, until $n/2$ elements are exchanged in the final step.

The partners in each stage are chosen so that the largest messages travel the fewest hops.

The cost is then $C_{2d-mesh} = 2 \lg P\alpha + (\frac{9}{4} - \frac{3}{2\sqrt{P}})n\delta$.

The next algorithm analyzed is the *buckets* algorithm. Suppose the P nodes are arranged in a $p \times p$ grid. The data is first divided into p buckets. In each of the first $p-1$ stages, one bucket is forwarded and combined along a ring formed by the columns in the grid. After the $p-1$ stages, each process has one bucket containing the reduction of the bucket for one column. These buckets are then further subdivided into p buckets and the process is repeated along the rows of the grid. The output data is now evenly scattered among the P processes. The algorithm then proceeds in the reverse order until every process has all the data.

The cost is: $C_{2d-mesh} = 4(\sqrt{P} - 1)\alpha + 2\frac{P}{P-1}n\delta$.

The authors also consider a hybrid of the recursive-halving and bidirectional exchange algorithms and a hybrid of the buckets and fan algorithms. The halving/exchange hybrid was shown theoretically and empirically to have the best performance on a 2-d 16x32 mesh.

In [31], a multi-ported bucket algorithm is presented for the reduce-scatter and allgather operations and implemented on a BlueGene/P. We discuss it further in Section 3.4.2, since it is quite similar to the multi-ported bucket algorithm we develop for mesh or torus systems.

Clos networks: In [32], an optimal broadcast algorithm is presented for Clos ("fat tree") topologies, with asymptotically optimal performance for several bandwidth-tapering approaches. It, too, uses a broadcast-tree approach,

with the distance between communicating nodes in each step chosen to minimize congestion. As expected, when the bisection bandwidth is proportional to the injection bandwidth, the time is bounded by $C_{Clos} = O(\lg P)n\delta$ for large messages.

In [33], an allreduce algorithm for tree topologies is presented, which is optimal with respect to the bandwidth cost. The input data on each of the P processes is partitioned into P sections and then forwarded along a one-dimensional embedded ring in $P-1$ steps. At the end of $P-1$ steps, each rank has $1/P$th of the output. Then an allgather operation is executed, using the same ring and $P-1$ more messages. By using an embedded ring, they ensure that there is no network congestion. The authors demonstrated good performance on 64 and 128-node systems. Their performance model does not consider message startup costs; for large systems with hundreds of thousands to millions of nodes, it is likely these overheads would become significant. The cost is $C_{Clos} = 2(P-1)\alpha + 2n\delta$.

Hierarchical networks: The Magpie system [34] provides collective communication operations optimized for wide-area networks composed of clusters. They organize their algorithms to minimize the use of the links between clusters. *E.g.*, reductions are performed within clusters, and then one result from each cluster is sent to the root node.

TMPI is an MPI implementation optimized for clusters of shared-memory nodes [25]. Processes located on the same node are mapped to threads and communicate through shared-memory. Collective algorithms are composed of intra-node and inter-node communication and organized such that only one process on each node participates in inter-node communication.

Hybrid networks: IBM's BlueGene series of supercomputers have two networks: a 3d torus used for point-to-point messages and some collective operations, and a tree network for other collective operations [20, 21]. The network switches also have special features, including provision for messages containing a "deposit bit" which are then broadcast to every node along a row or column in the torus. The switches in the tree network have support for integer reduction operations.

3.2.2 Generic-topology algorithms

The network details modeled in the work on generic-topology algorithms vary significantly.

Barnett, Van de Geijn *et al* present a clever approach for long broadcast operations [35]. The long broadcast message is scattered so that each process gets one small piece of the message. Then all the processes execute an all-gather operation. This method uses more of the available network bandwidth than the simpler binary-tree algorithm and achieves a lower bandwidth term in the two-parameter network model. This became known as the *Van de Geijn broadcast algorithm*.

Juurlink *et al* develop an optimal broadcast algorithm using a parallel-locality algorithm [36]. A latency function is introduced to model the latency between two nodes for a wide variety of networks. Their algorithm, however, does not model congestion, nor does it allow for breaking up a large broadcast message into smaller broadcast messages. The model is not empirically validated.

In [37], the classical binary-tree broadcast is given a highly-detailed treatment. The authors use the LogP model, which models latency, message startup cost, link bandwidth, and total system bandwidth [38]. This is a useful model, but it assumes bandwidth is uniform under every bisect pattern, *i.e.*, all bisect communication patterns suffer congestion equally. They develop an algorithm for generating detailed schedules for every message involved in a single-source and all-to-all broadcast. They extend the schedules to handle reduction and global-reduction problems. Their algorithm does allow for dividing a large broadcast message into smaller portions. They prove most of their algorithms to be optimal under the LogP model, in that not even the constant factors can be improved. The others are optimal within a factor of two. The model is not empirically validated.

One notable predecessor to MPI was CCL, which was designed by IBM in the early 1990s for their message-passing systems [39]. CCL and MPI provide equivalent collective communication operations. The authors of CCL use a three-parameter model to analyze the performance of their collective algorithms; this consists of the startup cost and bandwidth cost in our two-parameter model and a total system bandwidth cost, which is the sum of the bandwidth used by all the processes in an operation. This third term

can be used to roughly approximate congestion and is similar to the LogP congestion term.

For allgather, when the number of processes is not a power-of-2, CCL uses a variation of the recursive-doubling algorithm, which later became known as Bruck's algorithm. As with recursive-doubling, communication consists of $\lceil \lg P \rceil$ stages, and the size of the messages exchanged doubles in each stage. In the first stage, process i sends data to process $i + 1 \bmod P$ and receives data from $i - 1 \bmod P$. In the following stages, process i sends data to process $i + 2 \bmod P$, then $i + 4 \bmod P$, and so on, while receiving data from $i - 2 \bmod P$, then $i - 4 \bmod P$. Finally, each process must circularly rotate the data in the output buffer to restore the correct data ordering.

The algorithms used in OpenMPI [9] and MPICH [7, 8], broadly-speaking, are chosen as those that work best under our two-parameter model, ignoring network congestion. A third γ parameter is used to analyze reduction algorithms. The specific algorithms used in MPICH are described in [40].

MPICH uses recursive-doubling or Bruck's algorithm for the short-message allgather operation and a ring algorithm for large messages. Recursive-halving is used for reduce-scatter for smaller messages; a pairwise-exchange algorithm consisting of $P-1$ stages is used for long messages. In the pairwise-exchange algorithm, in stage s, process i sends data to process $i + s$ and receives data from process $i - s$. It has the same performance characteristics as the ring algorithm apart from the communication pattern, which is likelier to cause congestion.

The broadcast operation is implemented using the simple binary tree algorithm for short messages or Van De Geijn's algorithm for large messages, which consists of a scatter followed by an allgather.

Rabenseifner's algorithm [41] is used for the long allreduce operation. It consists of a reduce-scatter followed by an allgather. The bidirectional exchange algorithm described above is used for short messages or non-associative reduction operations.

Similarly, the long reduce operation is decomposed into a reduce-scatter operation followed by a gather, whereas the short or irregular reduce operation is implemented directly using a binary tree.

In [10], more detail on several collective algorithms used in MPICH is provided.

3.3 Performance of minimal algorithms

This work focuses on the performance of large allgather and reduce-scatter algorithms. Our work improves the performance of the large broadcast, reduce, and allreduce operations because they can be decomposed into combinations of allgather or reduce-scatter operations with other operations.

We now delve into how known allgather and reduce-scatter algorithms perform on torus and Clos networks.

In the allgather operation, each process i of P processes has an input vector X_i of length n. After the operation is complete, each process has the same copy of output vector Y of length nP. Vector Y consists of each X_i concatenated in rank order. *I.e.*, $Y = X_0X_1X_2 \cdots X_{P-1}$. Essentially, every process broadcasts its input vector X to all the other processes.

In the reduce-scatter operation, each process has an input vector X_i of length nP. Afterwards, each process has an output vector Y_i of length n. In the operation, blocks of n input elements from each X_i are combined using the reduction operator, and the result from block i is stored on the process with rank i. E.g., Y_0 will be formed from the first n elements of each X_i; Y_1 will be formed from the second n elements of each X_i and so on. Formally, $Y_i = \oplus_{p=0}^{P-1} X_p[in : (i+1)n-1]$, where $\oplus$ is the reduction operator (*e.g.*, sum, maximum, product, etc.).

Allgather can be easily implemented as P broadcast operations. Similarly, reduce-scatter can be implemented as P reduce operations, but more sophisticated algorithms deliver better performance.

These two algorithms are particularly important, because several other MPI operations can be efficiently implemented as combinations of allgather or reduce-scatter and other operations. In this work, we focus on allgather; the inverse of the communication pattern in each allgather algorithm is the same as the communication pattern in the corresponding reduce-scatter algorithm.

3.3.1 Non-topology-aware algorithms

Ring algorithm: One simple allgather algorithm is the ring algorithm, composed of $P-1$ stages. In each stage, process i sends data to process $i-1 \mod P$ and receives data from process $i+1 \mod P$. In the first stage, process i sends its input vector X_i and receives vector $X_{i+1 \mod P}$. In the

subsequent $P-2$ stages, each process sends the vector it received in the previous step. In stage s, where the stages are numbered from 0 to $P-2$, each process sends $X_{i+s \mod P}$ and receives $X_{i+s+1 \mod P}$.[1]

After P steps, each process has the full output vector $Y = X_0X_1X_2 \cdots X_{P-1}$.

The ring algorithm is easy to analyze: there are $P-1$ stages, so the message-startup cost is $P-1$. In a mesh, torus or Clos network, the ring algorithm will not cause any congestion, and the bandwidth term will be $n \cdot (P-1)$. The ring algorithm is optimal with respect to the bandwidth term on single-port networks.

The cost is:

$$C = (P-1)\alpha + n(P-1)\beta.$$

A similar ring algorithm exists for the reduce-scatter operation. In the first stage, each process i sends process $i-1$ the elements from its input vector corresponding to process $i+1$, *i.e.*, $X_i[(i+1)n : (i+2)n-1]$. It receives from process $i+1$ the data corresponding to process $i+2$ and combines this data (using the reduction operator) with the elements in X_i corresponding to process $i+2$. This proceeds similarly for the $s-2$ remaining stages.

The ring algorithms can also be run in the opposite direction.

Recursive halving/doubling algorithm: An important algorithm for allgather is known as the recursive-doubling algorithm. It is composed of $\lg P$ stages. In the first stage, process i exchanges its input vector X_i of n elements with the process whose rank only differs in the last bit. In the second stage, process i exchanges $2n$ elements with the process whose rank only differs in the second-last bit. In stage s (from 0 to $\lg P - 1$), process i exchanges $2^s n$ elements with process i `XOR` 2^s. It is named the recursive-doubling algorithm because the amount of data exchanged and the portion of the output vector Y that is filled doubles in each stage.

The corresponding algorithm for reduce-scatter is known as the recursive-halving algorithm. In the first stage, processes whose high bits differ exchange half of the input vector, in the second stage, processes whose second-highest bits differs exchange one fourth of the input vector, and so on.

If there is no congestion, the bandwidth term is given by the summation:

[1] Hereafter, arithmetic on process ranks is assumed to be circular on the number of processes P.

$\sum_{s=0}^{\lg P-1} 2^s n = n(P-1)$, and the cost is:

$$C = \lg P\alpha + n(P-1)\beta.$$

This algorithm is optimal with respect to the startup cost; data cannot be broadcast from one process to $P-1$ other processes in under $\lg P$ stages. However, on a $\mu < 1$ Clos network or a torus, there will be congestion.

Clos network: Let us consider first a Clos network where each terminal switch is connected to T nodes and T is a power-of-two. In the first few stages, where $2^s < T$, all communication will be between nodes connected to the same switch and there will be no congestion. In the remaining stages, all communication will be between nodes connected to different switches, and bandwidth will suffer by a factor of $1/\mu$. Thus,

$$\begin{aligned} C_{Clos} &= \lg P\alpha + n\delta\{\sum 1 + 2 + \cdots + T/2 \\ &+ 1/\mu \sum T + 2T + \cdots + P/2\} \\ &= \lg P\alpha + n\delta\{(T-1) + 1/\mu \cdot (P-T)\} \\ &\approx \lg P\alpha + (nP/\mu)\delta, \end{aligned}$$

where the approximation holds for $P \gg T$.

The effect of congestion reduces the performance of this algorithm by a factor of $1/\mu$ on the bandwidth term, which is the ratio of bisection to injection bandwidth.

Where T is not a multiple-of-two, even on the first stage, some communication will be between nodes on different switches. The portion will gradually increase until $2^s \geq T$. In the first stages, where $2^s < T$, 2^s of the T nodes attached to each switch will exchange messages with nodes connected to other switches. There will be congestion only when $2^s/T > \mu$, where the congestion factor will be $(2^s/T)/\mu$ rather than $1/\mu$. Thus, our approximation of the bandwidth term still holds, and is in fact, more accurate than when T is a power-of-two.

If T is a multiple-of-two, but not a power-of-two, the first stage where there is communication between switches will be where 2^s is not a factor of T.

3d torus: Let us consider the performance of the algorithm on a system composed of P nodes arranged in a $p \times p \times p$ 3d torus, where p is a power-

of-two. We assume that the nodes are numbered in increasing order in the X dimension, then the Y dimension, then the Z dimension.

In the first stage, nodes at a rank distance of one apart exchange input vectors; these nodes will be neighbors in the X dimension, so there will be no congestion. In the second stage, nodes at a rank distance of two apart exchange twice as much data as in the first step. If $X \geq 4$, these nodes will be two hops away in the X dimension, and there will be a congestion factor of two. In the third step, messages four times as large will travel four hops in the X dimension, with 4-fold congestion, and so on. In the second-last stage in each dimension, stage $\lg p - 1$, messages will travel $p/4$ hops and suffer $p/4$-way congestion. Since this is a torus, in the final stage in each dimension, stage $\lg p$, messages will travel $p/2$ hops, but will suffer only $p/4$-way congestion because in this stage only, the extra wrap-around link provided by the torus can be used.

Once we reach the stage where the rank distance between nodes is equal to X, neighbors in the Y dimension will exchange data, and so on.

If we revisit our model, we get a bandwidth term:

$$\begin{aligned}
& n(1+p+p^2)\{\sum 1 \cdot 1 + 2 \cdot 2 + \cdots + \\
& (p/4) \cdot (p/4) + (p/2) \cdot (p/4)\} \\
= \; & n(1+p+p^2)\{\sum_{i=0}^{\lg p - 1} 4^i - p^2/8\} \\
= \; & n(1+p+p^2)\{1/3(p^2-1) - p^2/8\} \\
= \; & n(1+p+p^2)(5/24p^2 - 1/3) \\
\approx \; & (5/24)nP^{4/3}
\end{aligned}$$

and a total cost:

$$C_{torus} \approx \lg P\alpha + (5/24)nP^{4/3}\delta.$$

Each factor 1, p, or p^2 represents the size of the first message exchanged in dimension X, Y, and Z, respectively. The first factor in each term in the summation represents the message size in the $\lg p$ stages in that dimension and the second term represents the congestion factor; note that the last two stages in each dimension have the same congestion factor.

The recursive-doubling algorithm is used for shorter messages in MPICH due to its lower startup cost term, and the ring algorithm is used for longer messages [7, 8].

3.3.2 Topology-aware algorithms

Bucket algorithm: The 3d bucket allgather algorithm is formed by executing the ring algorithm in each dimension in turn. *E.g.*, on a $P = p \times p \times p$ network, messages of size n circulate in the X dimension, then messages of size np circulate in the Y dimension, then messages of size np^2 circulate in the Z dimension.

This reduces the number of messages substantially compared to the ring algorithm and incurs a total cost:

$$C_{torus} = 3(\sqrt[3]{P} - 1)\alpha + n(P-1)\delta$$

Similarly, the 3d bucket reduce-scatter algorithm is formed by executing three ring reduce-scatter algorithms, starting with np^2 elements per message in the Z dimension, then np elements per message in the Y dimension, then n elements per message in the X dimension.

3.4 Non-minimal algorithms

In this section, we present several novel non-minimal recursive-doubling and bucket algorithms. They are non-minimal in the sense that there is redundant communication.

The emphasis is on the allgather operation; the data flow in the reduce-scatter operation is simply the reverse. allgather is a more attractive operation to use to explain our strategies, since the application of the reduction operator is an overhead that we do not address, and out techniques improve the communication performance alone.

We add a stage of communication before the allgather operation and after the reduce-scatter operation that enables us to reorder the stages of the algorithms while preserving correctness. It also enables the use of multiple ports in a multi-port torus network.

3.4.1 Recursive-doubling algorithm

The recursive-doubling allgather algorithm presented above can be described as a *distance-doubling*, recursive-doubling algorithm, because the rank distance between communicating processes doubles in every stage (1, 2, 4, ..., $P/2$) as does the size of the messages exchanged. Thus, the stages with the largest messages suffer from the worst congestion.

To resolve this, we propose a recursive-doubling, *distance-halving* allgather algorithm for Clos networks and a reordered recursive-doubling allgather algorithm for mesh or torus networks.

In the distance-halving algorithm, in each stage s, processes exchange $n2^s$ elements, as before. Process i exchanges data with process i `XOR` $P2^{-(s+1)}$. *I.e.*, in the first stage, processes exchange data with the process whose rank differs only in the highest bit, in the second stage, processes exchange data with the process whose rank differs only in the next-highest bit, and so on.

Clos networks: On a Clos network, the cost is then:

$$\begin{aligned} C_{Clos} &= \lg P\alpha + n\{1/\mu \sum 1 + 2 + \cdots + P/(2T) \\ &+ \sum P/T + 2P/T + \cdots + P/2\}\delta \\ &= \lg P\alpha + n\{(1/\mu)(P/T - 1) + (P - P/T)\}\delta \\ &\approx \lg P\alpha + nP\{1/(\mu T) + 1\}\delta \end{aligned}$$

From the first stage until the last stage where $P2^{-s+1} \geq T$, all messages must travel through multiple switches and incur congestion determined by the parameter μ. The largest messages in the final stages travel through only one switch with no congestion.

This analysis assumes P and T are powers-of-two. It is a close approximation for other values of T. If 2^a is the largest power of two that evenly divides T, then in the last a stages all the messages will travel through only one switch. For the next handful of stages, some pairs of processes will exchange messages through only one switch and others through several switches. Whether or not congestion occurs in the intermediate stages depends on the value of μ and T, but it is sharply curtailed in any case. There is no concise equation for the exact bandwidth term when T is not a power-of-two.

This algorithm reduces the bandwidth term in the cost from $(nP/\mu)\delta$ to

$nP\{1/(\mu T)+1\}\delta$, which is much closer to the optimal bandwidth term: $nP\delta$, however, the data is no longer in the correct order.

In the distance-doubling, recursive-doubling algorithm, each process receives a message in each stage containing elements in the output vector adjacent to the elements that the process already has. *E.g.*, process 0 starts with element 0, receives element 1 in stage 1, receives elements 2 and 3 in stage 2, 4-7 in stage 3, and so on. In each stage, each process simply concatenates the data it receives with the data it already has.

In the distance-halving, recursive-doubling algorithm, each process still concatenates the data it has with the data it receives in each stage, but the output ends up in the wrong order. The input vector X_i *s*hould start at offset ni in the output vector Y, but instead will end up at offset $n\tilde{i}$, where $\tilde{i}$ is defined as the bit-reverse of i.

To rectify this, we simply have processes i and $\tilde{i}$ exchange input vectors before the algorithm begins. Then each input vector ends up at the correct offset in the output vector.

We add this stage to the algorithm's cost:

$$\begin{aligned} C_{Clos} &= \lg P\alpha + nP\{1/(\mu T)+1\}\delta + \\ &\quad \alpha + (n/\mu)\delta \\ &\approx (\lg P + 1)\alpha + nP\{1/(\mu T)+1\}\delta. \end{aligned}$$

Each process sends and receives nP bytes instead of $n(P-1)$, a negligible difference.

For the corresponding recursive-halving, distance-doubling reduce-scatter algorithm, processes i and $\tilde{i}$ exchange output vectors.

We expect that even $\mu = 1$ networks might benefit from this technique. Network performance is known to suffer at high network loads, and full theoretical bisection bandwidth is not always delivered [4, 5]. Our algorithm, on any Clos network, will greatly lower the average network load.

For $\mu < 1$ networks, this algorithm delivers nearly the full injection bandwidth for reasonable values of μ, whereas the recursive-doubling, distance-doubling algorithm is restricted by the bisection bandwidth.

Torus networks: For mesh or torus networks, we *could* use the distance-halving allgather algorithm and expect an improvement, since the largest message in the last stage will not be delayed by congestion and the next

largest message will have a congestion factor of 2 instead of $\sqrt[3]{P}/4$. The distance-halving algorithm is really a quasi-topology-aware algorithm, since it will perform well for any network in which nodes are numbered rationally.

We can do better if we further reorder the stages. As before, let us assume that the network has $P = p \times p \times p$ nodes, where p is a power-of-two. In stages one through three, nodes $p/2$ hops apart in the X, Y, and Z dimensions, respectively, exchange data, suffering $p/4$-fold congestion. In the next three stages, processes $p/4$ hops apart exchange data, suffering $p/4$-fold congestion again. In the next group of three stages, congestion will be reduced to a factor of $p/8$, and will continue to halve for each subsequent group of three stages. In the last three stages, immediate neighbors exchange data with no congestion.

The bandwidth term is then:

$$\begin{aligned}
& (1+2+4)n\{\sum 1 \cdot p/4 + 8 \cdot p/4 + 64 \cdot p/8 + \cdots + P/8 \cdot 1\}\delta \\
\approx\ & 7n(p/2)\{\sum_{i=0}^{\lg p - 1} 4^i\}\delta \\
\approx\ & (7/6)np^3\delta \\
=\ & (7/6)nP\delta.
\end{aligned}$$

The first message exchanged in each dimensions is of size n times 1, 2, or 4. The first term in the summation reflects the size of the message; the first message exchanged in dimension X will be of size $1n$, the second message exchanged in dimension X will be of size $8n$, etc. The second term in the summation represents the congestion in that stage of the algorithm.

The data exchange to restore the correct ordering is more complex. Let us define a schedule, S, which dictates the distance between pairs of communicating processes in each stage. More precisely, S is a vector that specifies which bit differs between communicating pairs of processes in each stage.

For the distance-doubling, recursive-doubling algorithm, $S[s] = s$, since in stage s, process i and process i `XOR` 2^s exchange data. For the distance-halving algorithm, $S[s] = \lg P - 1 - s$. For the torus-reordered algorithm with XYZ process numbering,

$$
\begin{aligned}
S[s] \;=\; & \{3\lg p - 1, 2\lg p - 1, \lg p - 1, \\
& 3\lg p - 2, 2\lg p - 2, \lg p - 2, \\
& \ldots, \\
& 2\lg p, \lg p, 0\}.
\end{aligned}
$$

For example, for a 512-node system arranged as an 8x8x8 3d torus, $S[s]$ can range from 0 to 8 (since $\lg 512 - 1 = 8$). For distance-doubling, $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. For distance-halving, $S = \{8, 7, 6, 5, 4, 3, 2, 1, 0\}$. For the remapped algorithm, one optimal schedule is $S = \{8, 5, 2, 7, 4, 1, 6, 3, 0\}$. (The entries in S for each group of three consecutive stages can be permuted.)

This table shows the full schedule:

Stage	S[s]	Distance	Hops	Congestion	Size $\times n$
0	8	256	4 (Z)	2	1
1	5	32	4 (Y)	2	2
2	2	4	4 (X)	2	4
3	7	128	2 (Z)	2	8
4	4	16	2 (Y)	2	16
5	1	2	2 (X)	2	32
6	6	64	1 (Z)	1	64
7	3	8	1 (Y)	1	128
8	0	1	1 (X)	1	256

Note that the communication in the three stages with the largest messages occurs without congestion.

It is simple to handle non-cube tori: the first entries in the schedule must correspond to the largest dimensions. (In fact, only two of the eight torus configurations we evaluate later are cubes.)

We define a mapping function R, where $R(S, i)$ is the process which sends its input vector to process i before the allgather operation (or to which process i sends its output vector after the reduce-scatter operation). $R(S, i)$ permutes the bits in i according to S.

The process with rank i can be expressed using bit vector b_i, where $i = \sum_{r=1}^{\lceil \lg P \rceil} 2^{r-1} b_i[r]$. *E.g.*, if $P = 16$, $b_{11} = \{1, 1, 0, 1\}$. In the data-swapping stage, rank i then receives its data from rank $R(S, i) = \sum_{r=1}^{\lceil \lg P \rceil} 2^{r-1} b_i[S[r]]$.

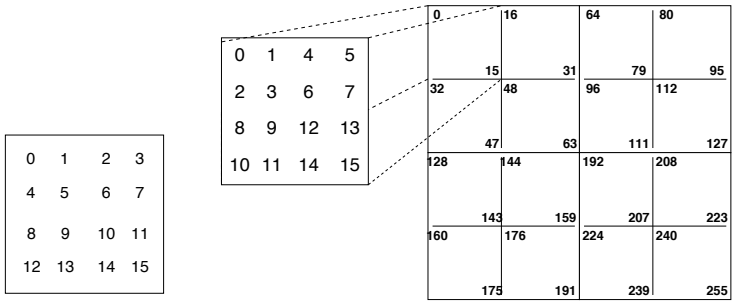

Figure 3.1: Row-major and Morton ordering on a 4x4 grid; Morton ordering recursively applied to a 16x16 grid [1].

The mapping function and schedule, when applied to a torus, are similar to the Morton ordering, which maps multidimensional integer coordinates to a one-dimensional integer coordinate while preserving multidimensional locality [1]. A one-dimensional Morton coordinate is formed by interleaving the bits of each of the k coordinates describing a point in a k-dimensional space. Morton orderings are typically used to form quad-trees or allow for efficient range-searching in a multi-dimensional space. In general, nearby coordinates in a k-dimensional space are likely to have a small Morton distance. Figure 3.1 illustrates a Morton ordering of a 4x4 and 16x16 grid. The left of the figure shows a traditional row-major ordering of a 4x4 grid, the middle of the figure shows a 4x4 Morton ordering, and the right portion of the figure illustrates how the Morton ordering is recursively applied.

In fact, if we reverse any optimal schedule and apply our remapping function, the result is a Morton order. Alternatively, our approach can be thought of as applying distance-halving on the Morton ordering; each process exchanges data with the process whose Morton ordering is $P/2$ away in the first stage, then $P/4$ in the second stage, and so on, with increasing communication locality as the algorithm progresses and the messages become larger.

We have investigated the exact traffic patterns in the data swapping phase for several different network sizes and found that none cause any congestion, but we cannot prove this. However, a 3d $p \times p \times p$ torus has $4p^2$ bisection bandwidth, therefore the bandwidth cost for this phase is bound by $nP/(4p^2)\delta = (n/4)\sqrt[3]{P}\delta$ from above and $n\delta$ below. The bandwidth cost of the extra stage is marginal compared to the bandwidth cost of the main algorithm.

Thus, the total cost is:

$$\begin{aligned} C_{torus} &\leq (\lg P + 1)\alpha + ((7/6)nP + (n/4)\sqrt[3]{P})\delta \\ &\approx (\lg P + 1)\alpha + (7/6)nP\delta. \end{aligned}$$

Non-power-of-two processors: MPICH uses Bruck's algorithm for the non-power-of-two allgather operation, but it used to use recursive-doubling with a workaround, which we borrow.

Recall that in stage s, process i exchanges data with the process $\hat{i}$ whose sth bit only differs from i. If that process does not exist, then in stage s, process i will do nothing. Instead, in stage $s+1$, it will send the data it *would* have sent in stage s to the process whose sth and $s+1$th bit differs from i; in other words, it sends data to the process whose $s+1$th bit differs from $\hat{i}$. This extension is straightforward to apply to our remapped recursive-doubling algorithm using R and R^{-1}. This can, in the worst case, double the number of stages.

3.4.2 Bucket algorithm

The bandwidth term in the bucket algorithm is optimal for the single-port model on a 3d torus. The number of stages, $3\sqrt[3]{P} - 1$, is more than the $\lg P$ stages of the recursive-doubling algorithm but much less than the $P-1$ stages of the 1d ring algorithm.

However, on a multi-port 3d torus network, each process can exchange messages with all six of its neighbors simultaneously. The bucket algorithm does not take advantage of this. Our approach extends the bucket algorithm to use all six ports.

In our algorithm, we run six bucket operations simultaneously in a way that avoids congestion. We label the six buckets XYZ^+, XYZ^-, YZX^+, YZX^-, ZXY^+, and ZXY^-. The + buckets run clockwise in each dimension and the - buckets run counterclockwise in each dimension. XYZ^+ and XYZ^- both circulate in the X dimension, then the Y dimension, then the Z dimension. Similarly, both YZX buckets circulate in the Y dimension, then the Z dimension, then the X dimension. We divide the input and output vectors into six sections, one for each bucket. Buckets 0 and 1 refer to the

XYZ buckets, 1 and 2 to the YZX buckets, and 2 and 3 to the ZXY buckets.

The cost is then:

$$\begin{aligned} C_{torus} &= 6 \cdot (3\sqrt[3]{P} - 1)\alpha + (n/6)n(P-1)\delta \\ &\approx 18\sqrt[3]{P}\alpha + (n/6)(P-1)\delta \end{aligned}$$

For large problems, the performance loss from the 6-fold increase in the number of messages is more than offset by the 6-fold decrease in the bandwidth term.

However, we face the familiar problem of data that is out-of-order. If the processes are numbered in XYZ order, then only a single XYZ bucket operation will be correct.

Let us show how the the 6-way bucket algorithm reorders the data. Suppose that each process i has 6 elements in its input vector $X_i = \{A_i, B_i, C_i, D_i, E_i, F_i\}$. The correct output would be $Y = \{A_0{-}F_0, A_1{-}F_1, \ldots\ A_{P-1}{-}F_{P-1}\}$. However, instead, in the first part of Y, we will get $Y[0 : n/6 - 1] = \{A_0, A_1, \ldots, A_{P-1}\}$. In the second part, we will get $Y[n/6 : n/3 - 1] = \{B_0, B_1, \ldots, B_{P-1}\}$. In the third part (from YZX^+), we will get $Y[n/3 : n/2 - 1] = \{C_0, C_p, C_{2p}, \ldots, C_{P-1}\}$, since the output will be in YZX order. The data in the remaining three sections will be similarly perturbed.

To solve this, we number each of the six sections in each input vector X_i. Section j in input X_i is assigned *section identifier* $6i + j$. In the correct output vector, all the input sections are arranged in section id order. In total, we have $6P$ sections. Sections whose id is between 0 and $P - 1$ must be placed in the first bucket, XYZ^+; sections whose id is between P and $2P - 1$ must be placed in the second bucket, XYZ^-, and so on.

Next is the question of *where* to place each section in the correct bucket. Section s must be at offset $s \mod P$ in the output of the corresponding bucket. For the two XYZ buckets, this is achieved by placing section s in the corresponding input bucket on process $s \mod P$.

For the YZX buckets, section s must be placed in one of the YZX buckets on the process whose rank *would be* $s \mod P$ if the processes were numbered in YZX order. Since they are not, we circularly rotate the bits in $s \mod P$ to find the rank of the process which have section s in its input buffer.

Consider an $8 \times 8 \times 8$ torus again. The segment id of the third segment from process 200 is $s = 200 \cdot 6 + 3 = 1203$. $\lfloor 1203/P \rfloor = 2$, therefore this segment will be in bucket 2: YZX^+. Its offset within YZX^+ is $1203 \mod P = 179$. This segment must start from the input vector of the process whose YZX order is 179. We then take the 9-bit quantity 179 and rotate it clockwise 3 bits to put it in XYZ order. The rotated offset is 410. Therefore the segment whose id is 1203 should be placed in $X_{410}[YZX^+]$.

For segments in the ZXY buckets, we rotate $s \mod P$ 6-bits clockwise (or 3-bits counter-clockwise).

In general, for each of the 6 sections in each input vector X_i, we assign a section id: $s = 6i + j$, where j is a value from 0 to 5. This section should be placed in bucket $\lfloor s/P \rfloor$. We then find the quantity $s \mod P$. If the section maps to either XYZ bucket, then this section should be sent to process $s \mod P$. If the section is in either YZX bucket, we rotate $s \mod P$ by $\lg p$ bits clockwise and send the data to process $(s \mod P) >> \lg p$.[2] If the section is in either ZXY bucket, we rotate $s \mod P$ by $\lg p$ bits counter-clockwise and send the data to $(s \mod P) << \lg p$.

The correct remapping may be more simply determined empirically: a correct method is to run the 6-bucket algorithm where each process sets its input to $X_i = \{(i,0),(i,1),(i,2),(i,3),(i,4),(i,5)\}$. Then each process inspects the tuples in $Y[6i : 6i+5]$; the first member in tuple $Y[6i+j]$ is the process that segment j from X_i should be sent to; the second member is which bucket that segment should be placed in. This mapping can be derived once for the system and stored in a configuration file.

Including the data-shuffling stage, the cost is:

$$\begin{aligned} C_{torus} &= (18\sqrt[3]{P} + 6)\alpha + ((n/6)(P-1) + n)\delta \\ &\approx 18\sqrt[3]{P}\alpha + (n/6)(P-1)\delta. \end{aligned}$$

The additional data movement adds negligible overhead.

Non-cubic torus networks are easily handled. Each of the six bucket operations waits until all the others have finished circulating in each dimension before moving on to circulate in the next dimension. We found that performance suffered greatly if we allowed multiple bucket operations to compete for the same links.

[2] $>>$ and $<<$ are the cyclical clockwise and counter-clockwise bit rotation operators.

Performance in the first two stages will be limited by the pair of buckets circulating on the largest dimension. In the last stage, in which the messages are much larger than in the first two stages, the buckets will have more equal performance. The buckets circulating along the longest dimension will have the smallest messages, and the buckets circulating along the shortest dimension will have the largest messages.

Consider a network where $X < Y < Z$. In the last stage, the slowest buckets will be the XYZ buckets and the quickest the YZX buckets. XYZ buckets will circulate $Z-1$ messages of XYn bytes, for a total of $XY(Z-1)n$ bytes. The YZX buckets will circulate $X-1$ messages of YZ bytes, for a total of $(X-1)YZ$ bytes. On large systems, the difference is not important.

Messages whose size is not a multiple of six are also simple to handle. We use a similar technique to that used in [42] for the irregular `MPI_Allgatherv` problem, which is like allgather, except that each process can broadcast a differently-sized message. We round-up the input size to the next multiple of six bytes and shift the data from higher processes to lower processes. Processes whose rank assignment is near P may have empty input buffers. This will involve each process sending at most two messages or receiving up to six messages, but typically no process will receive more than three messages.

For 3d meshes, or partitions of 3d tori, the 6-way bucket algorithm will cause a congestion load of 2 on each link (since there is no wrap-around link). The 3-way bucket algorithm would then be preferred for its lower startup cost.

Further, note that while our presentation describes the algorithm on a 3d cube torus, it is trivially extended to torus or mesh networks with any number of dimensions, which may be equally-sized or not.

Known multi-port algorithms: A variation of this algorithm was presented in [31]. Instead of adding an extra stage of communication to restore the correct data ordering, they reorder the data after the allgather operation or before the reduce-scatter operation. We also considered three alternatives to the extra stage: using MPI types to send and receive messages with non-contiguous memory addresses, packing the data into temporary buffers before sending messages and unpacking the data from temporary buffers after receiving messages, and reordering the data in-memory at the end. We found all three to have poor performance. The presentation in [31] is difficult to

follow, since they extend the bucket algorithm but present the ring algorithm instead in their description of the base bucket algorithm. The description of the handling of non-multiple-of-6-sized vectors is very sketchy. They explored multiple methods of multithreading the operation of the 6-way bucket algorithm; we do not. We will compare our performance results with theirs in the following section.

3.4.3 Recursive-doubling algorithm with irregular partitions

Handling noncontiguous blocks of processors is more challenging than handling contiguous blocks. Here we provide a method to adapt the recursive-doubling allgather or recursive-halving reduce-scatter algorithms to irregular groups of processors. This is important for handling sparse collective operations [43].

The solution involves first finding one Hamiltonian path through all the processors in the system. The authors of [44] present an algorithm to find k edge-disjoint Hamiltonian paths in a k-d mesh network which is similar to the algorithm given in [45], which finds k Hamiltonian paths in a k-d torus network. On a Clos network, we can simply create a ring using a depth-first traversal.

We then simply delete all the processors from the Hamiltonian path that are not in the irregular partition.

In an immediate-neighbor pattern in a Hamiltonian path, all can communicate without any congestion if messages are routed along the Hamiltonian path. Neighbors two apart can communicate with a congestion factor of two. Neighbors four apart can communicate with a congestion factor of four, and so on.

The congestion can be less if the routing algorithm considers other paths or can be more if the irregular partition is unfavorable, since no realistic routing algorithm will use Hamiltonian-path routing. Let us assume for now that Hamiltonian-path routing is used.

We then run the recursive-doubling, distance-halving algorithm on the rank of each node in the Hamiltonian path. On a mesh or torus network with ideal routing, performance will be at least as good as that delivered by the recursive-doubling, distance-halving algorithm on a contiguous block.

On a Clos network with ideal routing, this would result in performance at least as good as that in the contiguous-partition algorithm for the first, bisection bandwidth-bound stages of the algorithm. In the later stages, the communication in the contiguous algorithm all takes place between processors attached to the same switch, congestion-free, if the number of processors attached to each stage is a power of two. In this non-contiguous algorithm, the performance in the final stages will be more similar to the non-power-of-two case for the contiguous algorithm, which we explained in Section 3.4.1.

Performance will be limited by the switch that has the most member processors in the irregular partition. If any of the processors attached to the busiest switch are not in the partition, then performance will be better than in the regular-partition algorithm, since the same amount of upstream bandwidth is shared among fewer processors.

Of course, real message routing algorithms are not going to be aware of the Hamiltonian path, and there may be more congestion caused by non-member processors communicating.

We have to modify the data-shuffling procedure. For allgather, the process with rank i in the non-contiguous partition must send its input data to the process at offset $\tilde{i}$ in the Hamiltonian path, where $\tilde{i}$ is defined as the bit-reverse of i.

Multi-port recursive-doubling operation: On contiguous partitions on torus or mesh networks, the best algorithm, in terms of the bandwidth cost, is the multi-port bucket operation.

We can adapt the non-contiguous recursive-doubling/distance-halving algorithm described above for multi-port operation on mesh or torus networks. Since there are three edge-disjoint Hamiltonian paths, we can run three recursive-doubling operations in parallel by dividing the input message into three parts. The process of reordering the input data is similar to that in the 3-bucket or 6-bucket algorithm.

We could use the multi-port recursive-doubling algorithm on regular partitions as well. In Section 3.6, we discuss our unfavorable experience with a hybrid of the triple recursive-doubling and 6-bucket algorithm. This is why we did not pursue a multi-ported recursive-doubling algorithm for contiguous partitions.

3.4.4 Other operations

Long broadcast operations in MPICH use the Van de Geijn algorithm, in which the operation is composed of a scatter operation followed by an allgather operation [35]. Similarly, the long reduce operation can be composed of a reduce-scatter, followed by a gather, and the long allreduce can be broken into an reduce-scatter call followed by allgather. Our algorithms can accelerate all of these operations. Further, the data reordering stages can often be made to cancel out; this happens naturally for allreduce. With simple changes, the data reordering step can be folded into the gather or scatter operation without any extra overhead. In fact, MPICH already uses a recursive-halving, distance-doubling reduce-scatter as the basis for the long reduce and allreduce operations, with no explanation given.

3.4.5 Alternatives to redundant communication

Our algorithms have the drawback of redundant communication compared to the corresponding minimal algorithms. There are several alternatives to extra communication. We could reorder the misordered data in memory after the allgather operation or before the reduce-scatter operation; we could use MPI types to exchange messages formed from non-contiguous data; and, finally, we could simply reorder the process-rank assignments to begin with so that the communication patterns of the minimal algorithms do not cause congestion. A bit-reversed Morton ordering would accomplish this for recursive-doubling.

The first two alternatives we evaluate in Section 3.5. The third alternative is unattractive because it would only work for the recursive-doubling algorithm (and not the multi-port bucket algorithm), it would make efficient point-to-point message-passing difficult, such as for common halo operations, and the performance of collective algorithms optimized for simple rank orderings would suffer.

3.5 Evaluation

All experiments were performed on the Intrepid system, a 40k-node Blue Gene/P, at Argonne National Labs. Partitions of 512 nodes or more form 3d

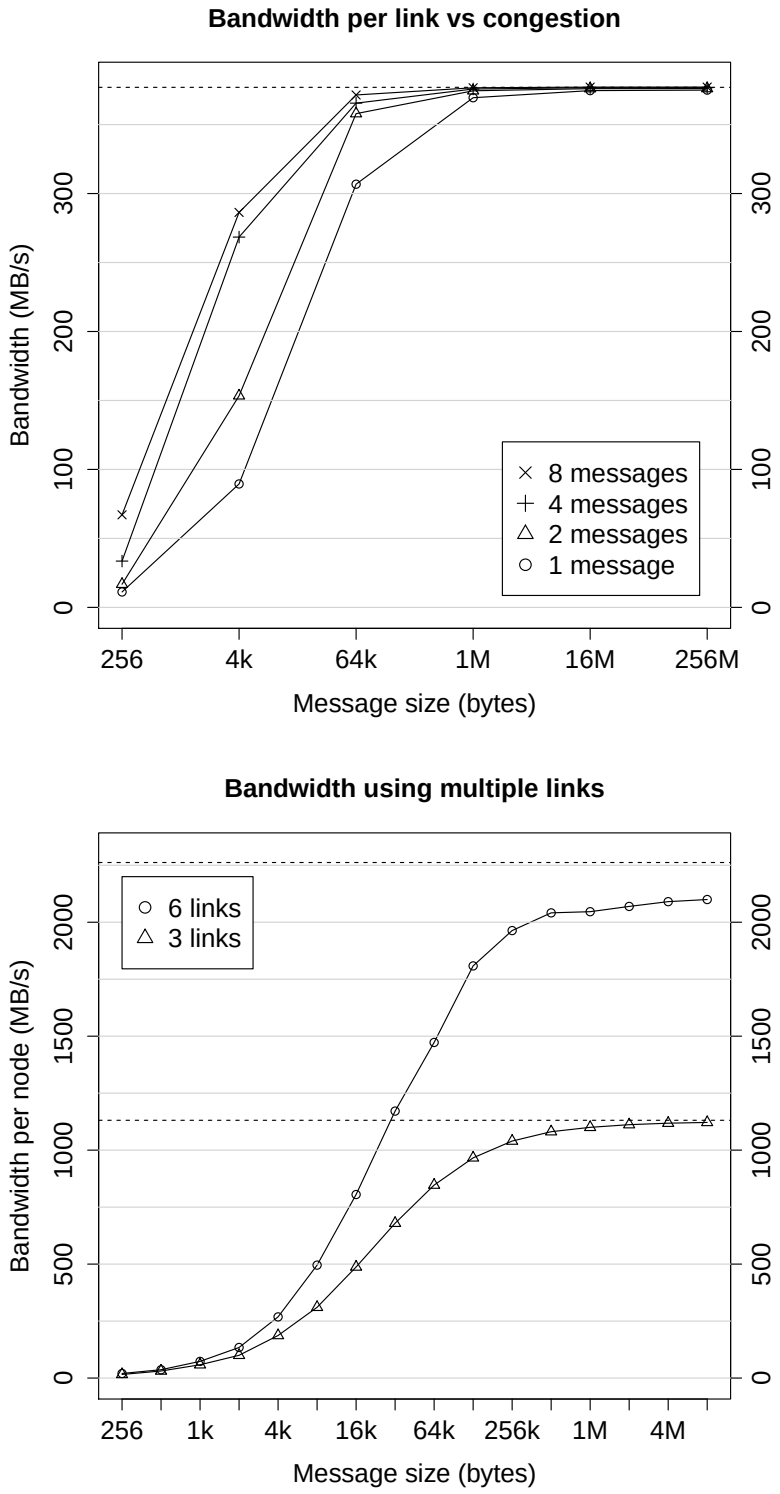

Figure 3.2: Link bandwidth vs. message size with congestion; Node bandwidth for multi-port communication

tori. Smaller partitions form 3d meshes. The Blue Gene/P has a torus network for point-to-point messages and some collective-communication operations and a tree network used for other collective-communication operations. Both networks have special features to accelerate collective communication. The torus network has a "deposit bit" feature in which a message can be put on the network once and every node along a line in the torus can receive it. This is particularly helpful for broadcast operations.

The 512-node allocation forms an 8x8x8-node cube. The 32k-node allocations forms a 32x32x32-node cube. The allocations in between are non-cubic. The Z dimension doubles as the size of the allocation doubles until it reaches 32 nodes, then the Y dimension doubles, then the X dimension. We run one MPI process on each node. The system provides a simple interface for finding the size of each dimension in an allocation and the coordinates of each process in the allocation. By default, processes are numbered in XYZ order.

Each node is comprised of four 850 MHz PowerPC 450 microprocessors and a switch. Each switch has 3 bidirectional links on the tree network of 850 MB/s per switch per direction and 6 bidirectional links on the torus network of 425 MB/s per switch per direction. The minimum packet size is 32 bytes, with 16 bytes of payload data and 16 bytes of header data. The maximum size is 256 bytes with 240 bytes of payload data. After accounting for the overhead in the packet headers and the acknowledgement packets, the maximum deliverable bandwidth per link is 88% of the raw bandwidth. According to IBM, with 6-way bidirectional communication, 93% of that is the peak that can be delivered, *i.e.*, 425 MB/s$\times 6 \times 2 \times .88 \times .93 = 4.18$ GB/s [46]. In our data, we report the one-way bandwidth into or out of a node. The bandwidth calculations count the time but not the data exchanged in the extra stage for the reordered recursive-doubling algorithms. *I.e.*, we report $n(P-1)/t$, where n is the input message size on each process, P is the number of processes, and t is the amount of time each operation takes.

The output of each combination of algorithm, size, and number of processes was verified for correctness. This includes padding messages to multiples of 3 or 6 bytes for the 3-bucket or 6-bucket algorithms. We use the IBM XLC compiler with the `-O4` level of optimization.

For each data point, we report the minimum time from four runs. We do this to eliminate infrequent outliers that would distort the data if we averaged several runs. Apart from a handful of outliers, there was very

little variation in the data for each configuration. *E.g.*, for the 32k-node partition with 16 kB input messages/process, the run times for the 6-bucket algorithm were 260.797, 260.789, 260.843, and 260.804 milliseconds. For 32 kB input messages, the run times were 517.573, 517.576, 517.598, and 560.217 milliseconds. The last time is an example of an outlier due to some interference on one of the 32k nodes in the allocation. None of the algorithms are particularly sensitive or insensitive to system noise. The effect of noise has been analyzed in great detail in [47].

3.5.1 Bandwidth under congestion or multi-port communication

We first investigated whether our performance model was accurate enough to justify development of our algorithms. In particular, we were interested in whether or not our congestion model for the recursive-doubling algorithms was accurate. The first plot in Figure 3.2 shows the link bandwidth versus message size as we vary the congestion (or the number of messages competing for the same link).

This data is from a 2048-node allocation (8x8x32 nodes). We examined all the patterns under the recursive-doubling communication patterns (where each process exchanges data with the process whose rank differs in only one bit). The maximum congestion on this allocation occurs along the Z axis, when processes whose Z coordinates differ by 16 or 32 exchange data. In either case, the congestion is 8-fold. The plot only shows data collected when the exchange occurs along the Z axis, but we found that the numbers were insensitive to the axis or the partition size.

We first see that, indeed, the peak bandwidth is 88% of the raw bandwidth: 375 MB/s. This is where we place the dashed line. Second, we see that, for large messages, the per-message bandwidth scales inversely with congestion (since the per-message bandwidth times the amount of congestion is a constant 375 MB/s). For smaller messages, congestion has less of an effect, but still harms performance. The fixed startup cost of a message explains part of the discrepancy. 375 MB/s is an upper bound on the bandwidth of any one-port algorithm.

We then ran an experiment to ascertain the bandwidth in the nearest-

neighbor communication pattern, where each node either sends data to 3 neighbors and receives data from 3 other neighbors, or where each node sends and receives data from all 6 neighbors. The second plot in Figure 3.2 shows the results. The dashed lines are at three and six times the maximum single-port bandwidth. For large messages, the 3-neighbor pattern achieves very nearly $3 \cdot 375\text{MB/s} = 1125\text{MB/s}$. For large messages, the 6-neighbor pattern achieves 2100 MB/s, which is 93% of six times the deliverable link bandwidth of 375 MB/s.

In the following algorithm-performance figures, the message size refers to the size of the input vector for the allgather operation or the output vector for the reduce-scatter operation, *i.e.*, the value of n from Section 3.4. The size of the output allgather vector or input reduce-scatter vector is this value times the number of processes, *i.e.*, $n \cdot P$.

3.5.2 Allgather performance

Figure 3.3 shows the performance of the native and one-port allgather algorithms as we scale the size of the problem. The upper plot shows the performance of an 8x8x8 512-node 3d torus. The recursive-doubling, distance-doubling algorithm (rd doubling) is the better of the two non-topology-aware algorithms for smaller messages, due to the lower message count of the recursive-doubling algorithms, and the ring algorithm is better for larger messages, due to the absence of congestion in the ring algorithm. The quasi-topology-aware recursive-doubling, distance-halving algorithm (rd halving), with one more stage of communication, beats the recursive-doubling, distance-halving algorithm for all message sizes.

Of the topology-aware algorithms, the bucket algorithm is better for larger messages and the optimally-scheduled recursive-doubling algorithm (rd optimal) is better for smaller messages. This is expected, since the recursive-doubling algorithm has fewer stages but a larger bandwidth term. The native algorithm performs best for smaller messages and next-best for larger messages on the 512-node allocation. The native algorithm data was collected using the `MPI_COMM_WORLD` communicator; performance using a copy or sub-partition of `MPI_COMM_WORLD` can be worse, as we will see later.

The optimally-scheduled recursive-doubling algorithm significantly reduces

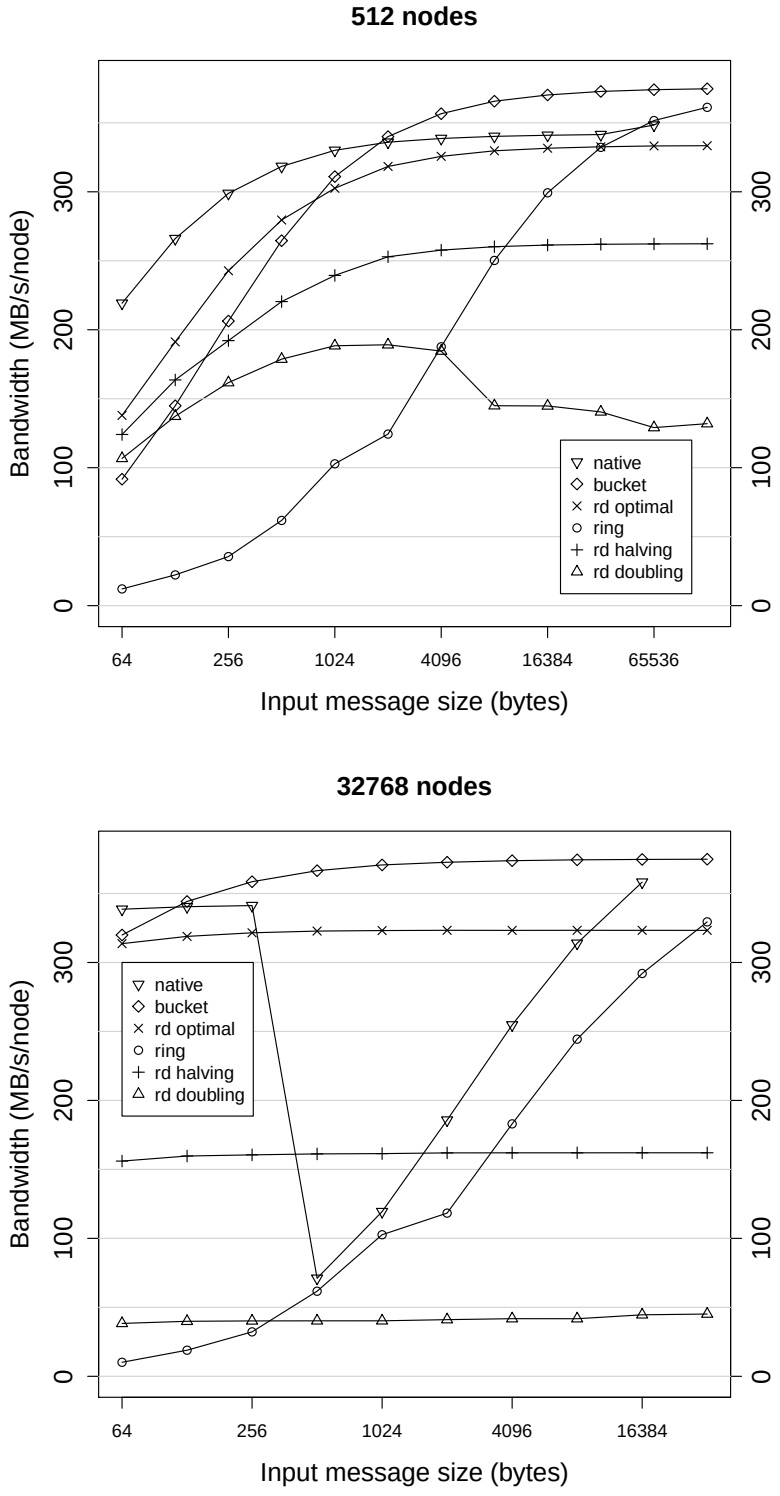

Figure 3.3: Performance of single-port allgather algorithms *vs* message size.

but does not eliminate congestion, whereas the bucket algorithm does not suffer from congestion at all. The bucket algorithm reaches the upper bound of 375 MB/s, whereas the best recursive-doubling algorithm reaches 333 MB/s. Our performance model of the optimal recursive-doubling algorithm has a bandwidth congestion factor of 7/6, so we would predict the performance to be $375/(7/6) = 321$ MB/s. We calculated this figure by making some approximations that introduce little error for large torus networks. In particular, it ignores the benefit of the wrap-around links. When this is taken into account, on a small 8x8x8 torus, the corrected congestion factor is 1.09, so our predicted performance would be 343 MB/s. In Section 3.5.6, we present a detailed comparison of our model with measured data.

The second plot in Figure 3.3 shows the performance of the native and single-port algorithms on a 32x32x32 32k-node 3d torus. Compared to the smaller 512-node torus, the non-congestion-free algorithms will face more congestion and all the algorithms will have more stages of communication. For these reasons, we see that it is far more important to use topology-aware algorithms to get good performance. Due to memory constraints, we cannot run the algorithms with 64 kB or 128 kB input messages on the 32k-node partition. Again, the bucket algorithm delivers 375 MB/s. The optimal recursive-doubling algorithm delivers up to 323 MB/s, which is very close to our prediction of 321 MB/s. The MPI library appears to prematurely change algorithms for 512-byte or larger messages and performance suffers.

Figure 3.4 shows the performance when we keep the problem size constant and vary the number of processors. We see that for large messages, performance is relatively invariant with the size of the torus. For larger messages, the bandwidth term in the performance model dwarfs the message startup-cost term. On the other hand, we see that for small messages, message startup time is important.

Figures 3.5 and 3.6 shows the performance of the multi-port algorithms and the native BlueGene/P allgather algorithm. The 3-bucket algorithm delivers up to 1120 MB/s into each node, or 99.5% of the achievable link bandwidth of three ports. The 6-bucket algorithm delivers up to 2075 MB/s, which is 99% of the achievable link bandwidth of 6 ports.

There are two native allgather algorithms used on the BlueGene/P [21]: allgather via randomized all-to-all, wherein each process swaps data with every other process in $P - 1$ steps; and allgather via broadcast where each process

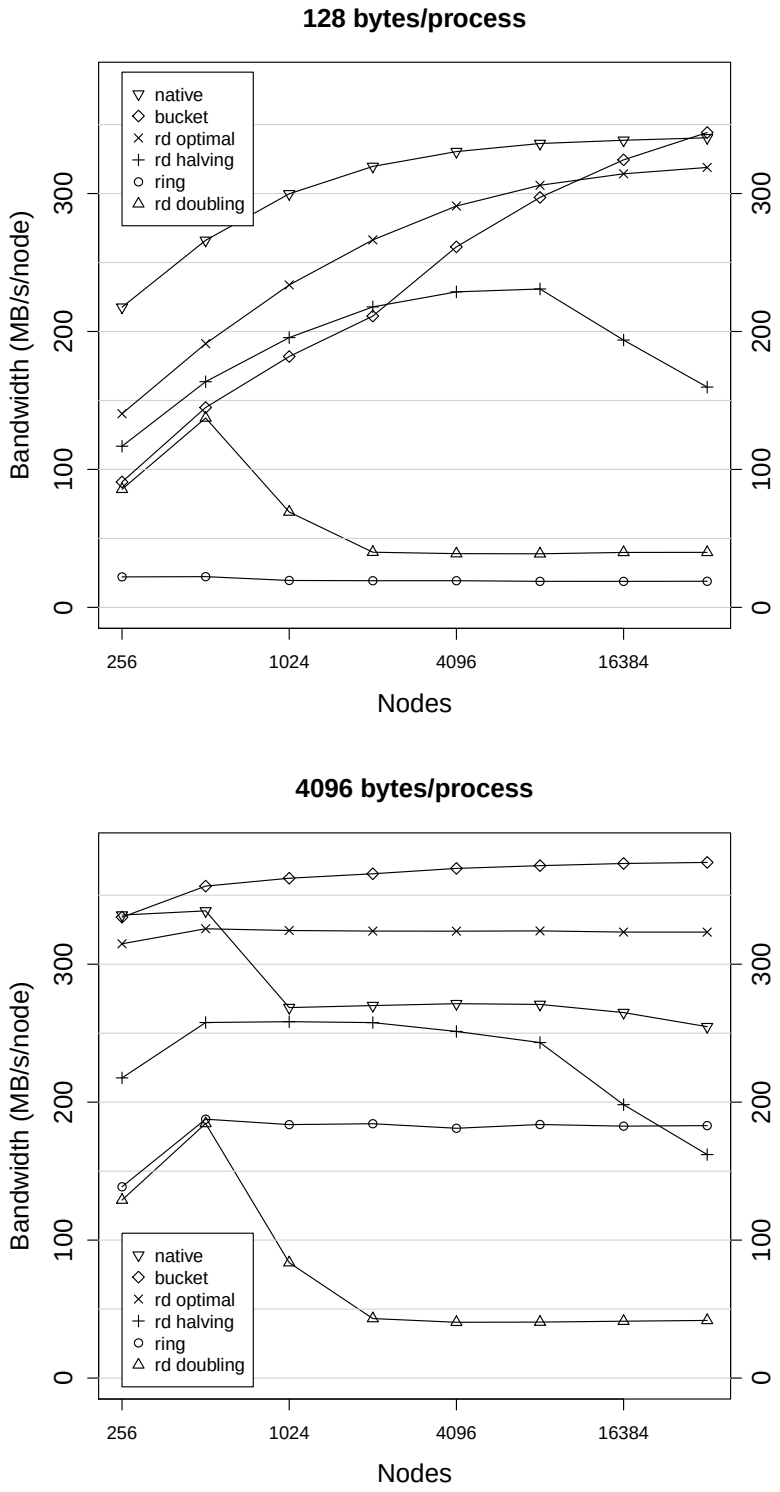

Figure 3.4: Performance of single-port allgather algorithms *vs* number of processors.

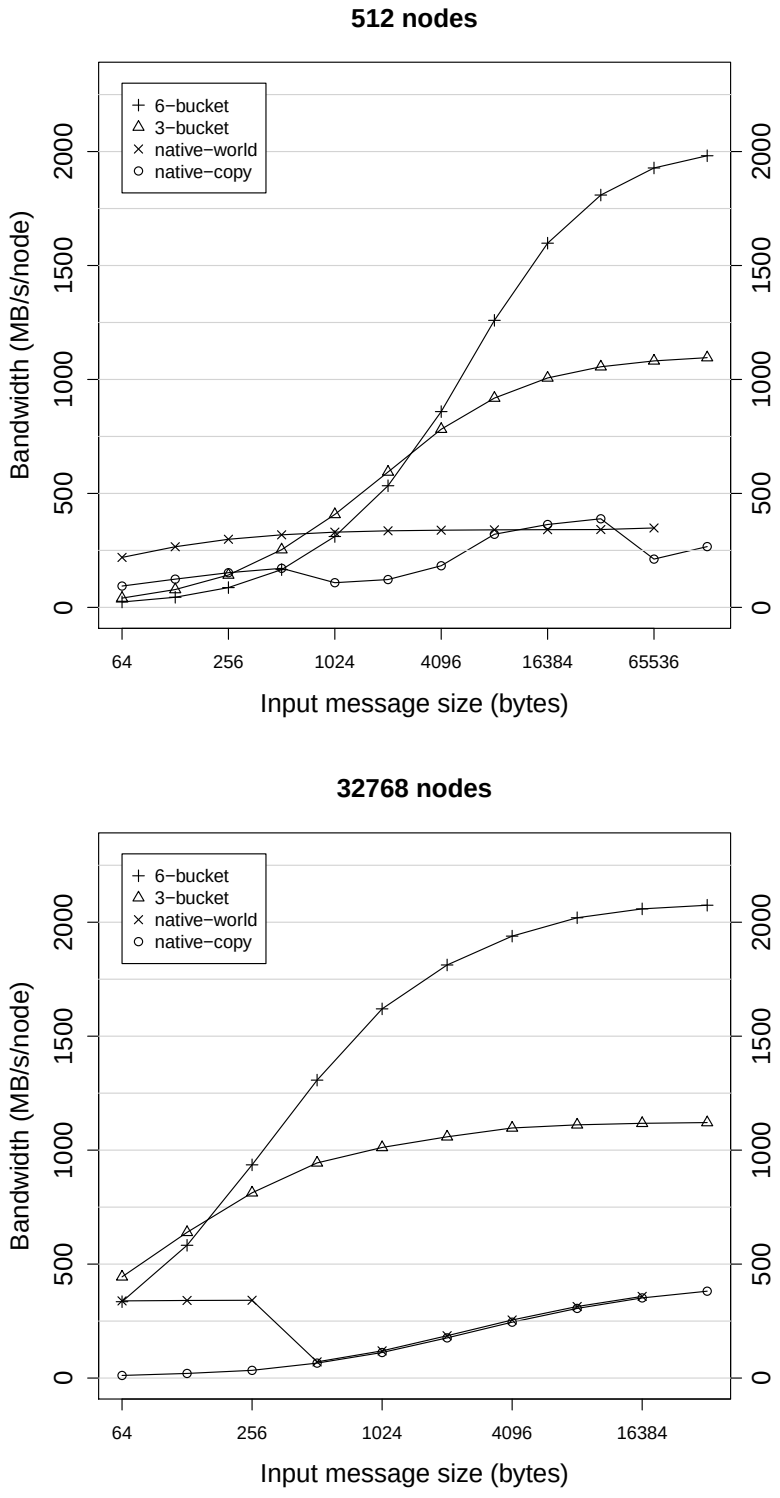

Figure 3.5: Performance of native and multi-port allgather algorithms *vs* message size.

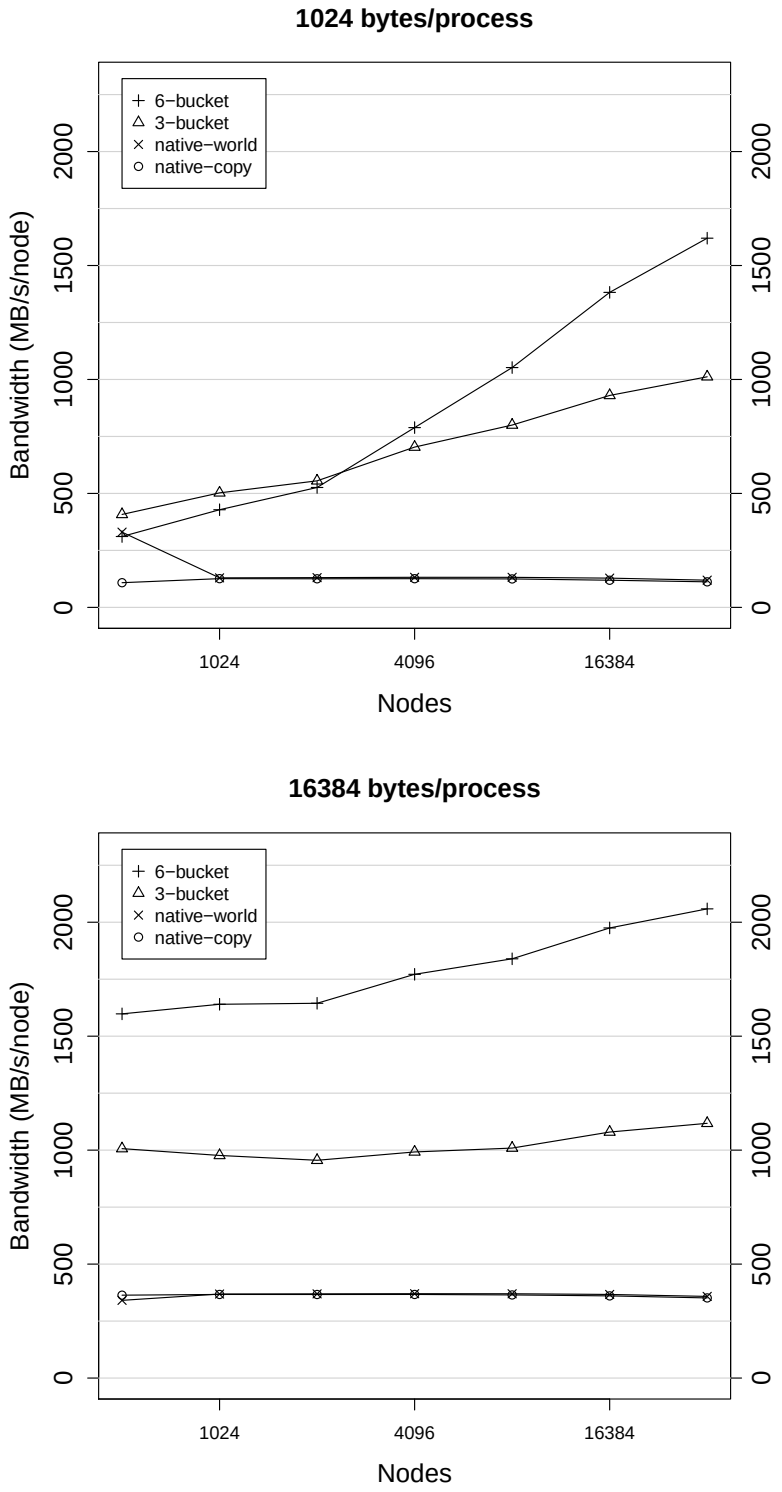

Figure 3.6: Performance of native and multi-port allgather algorithms *vs* number of processors.

in turn broadcasts its message. The second algorithm can be performed on the tree network or the torus network using line-broadcast support. It is not known how the native MPI library chooses which algorithm to use. In these figures, we provide data for the native algorithm running on `MPI_COMM_WORLD` and on a copy. The Blue Gene/P MPI library uses different algorithms for operations on the entire partition, contiguous subpartitions, or noncontiguous subpartitions. We see that the copy sometimes matches the performance of the native algorithm and sometimes performs worse. Over nearly every combination of partition size and message size, our algorithms perform far better than the native algorithm.

Figure 3.7 shows the amount of time spent in the additional data-exchange steps for the optimally-remapped recursive-doubling algorithm (rd optimal) and the 6-way bucket algorithm. The bucket algorithms have to shuffle data once for non-multiple-of-6-sized messages and a second time to restore the correct ordering. Moreover, in the order-restoring shuffle, each process typically exchanges data with 12 other processes. In contrast, there is only one exchange of data in the optimally-remapped recursive-doubling algorithm and each process sends data to one process and receives data from another. For this reason, with 512 nodes, the recursive-doubling algorithms have an overhead of 2.3% or less, whereas the 6-way bucket algorithm has an overhead of up to 7%. For 32k nodes, the overhead is much less, since the total amount of communication is proportional to the number of nodes and the communication in the data shuffling phases is not.

Alternatives to redundant communication: We explored several alternatives to having an additional stage of communication. In Figure 3.8, we show the performance of two alternatives to the extra stage for the recursive-doubling, distance-halving allgather operation running on 512 nodes. We could reorder the data in memory after the recursive-doubling operation is complete (shown as "in-memory shuffle" in the figure). We load data from bit-reversed offsets in a temporary misordered output vector and store the data in sequential order in the correctly-ordered output vector; this was quicker than the reverse. We also tried using MPI types to read and write strided, non-contiguous data. Both proved slower than our method with an extra stage of communication (shown as "swap"). We chose to use the distance-halving algorithm as the basis for this experiment, since the system is likely to deliver better performance exchanging strided, non-contiguous

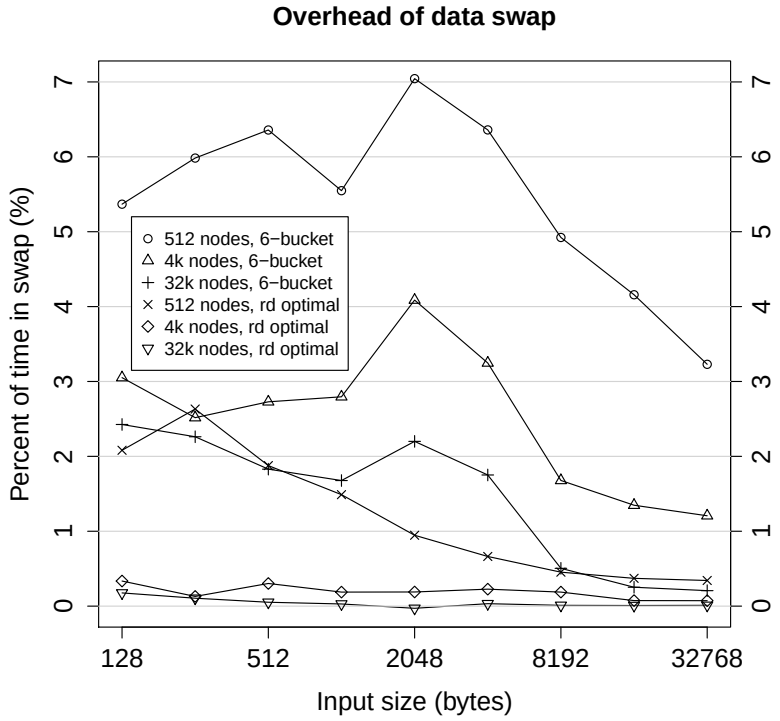

Figure 3.7: Overhead of input data shuffle stage.

messages than non-contiguous messages with a more complex memory layout.

According to the vendor, the streaming load bandwidth from each core to memory is 4.6 bytes/cycle, which is 3.9 GB/second [48]. The streaming store bandwidth is 5.6 bytes/cycle, or 4.75 GB/second. In [49], the streaming copy bandwidth on Intrepid for large buffers was measured to be 3.85 GB/second. This figure represents the sum of the load and store bandwidth, and is, in fact, lower than the peak deliverable bidirectional 6-port network bandwidth of 4.18 GB/second. The copy bandwidth on a random memory-access benchmark was 44 MB/s. From this, we conclude that unless memory bandwidth dwarfs network bandwidth, our redundant communication method is superior to alternatives.

3.5.3 Allgather performance on subpartitions

As mentioned above, partitions of 512 or more nodes on Intrepid form tori. We decided to investigate how our algorithms perform on subpartitions,

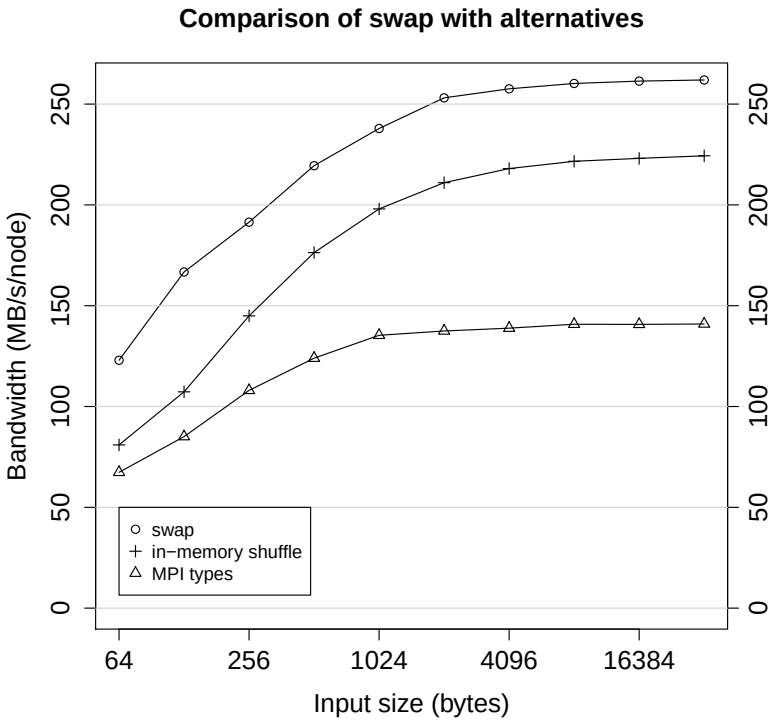

Figure 3.8: Alternatives to input data shuffle.

which form meshes. Figure 3.9 shows the performance of subpartitions of a 32k-node partition. All dimensions are powers of two, but we make the subpartitions as cubic as possible. Native algorithm data in these figures is always from a copy or subpartition of `MPI_COMM_WORLD`.

Note that while the 16k subpartition is toroidal in two dimensions and the 8k subpartition is toroidal in one dimension, performance is still limited by the one non-toroidal dimension. A 4-bucket algorithm when there is one toroidal dimension or a 5-bucket algorithm when there are two toroidal dimensions might perform better. Also note that the 6-bucket algorithm performs worse than the 3-bucket algorithm. The 3-bucket algorithm will operate without congestion, whereas the 6-bucket algorithm will operate with a congestion factor of two on every link. In Section 3.5.1, we found that congestion did not cause link bandwidth to suffer for single-port communication; in this experiment, with multi-port communication, congestion appears to have a more performance-degrading effect. If the effect were solely due to the larger number of messages, we would expect the performance divergence between the 3-bucket algorithm and 6-bucket algorithm for smaller messages

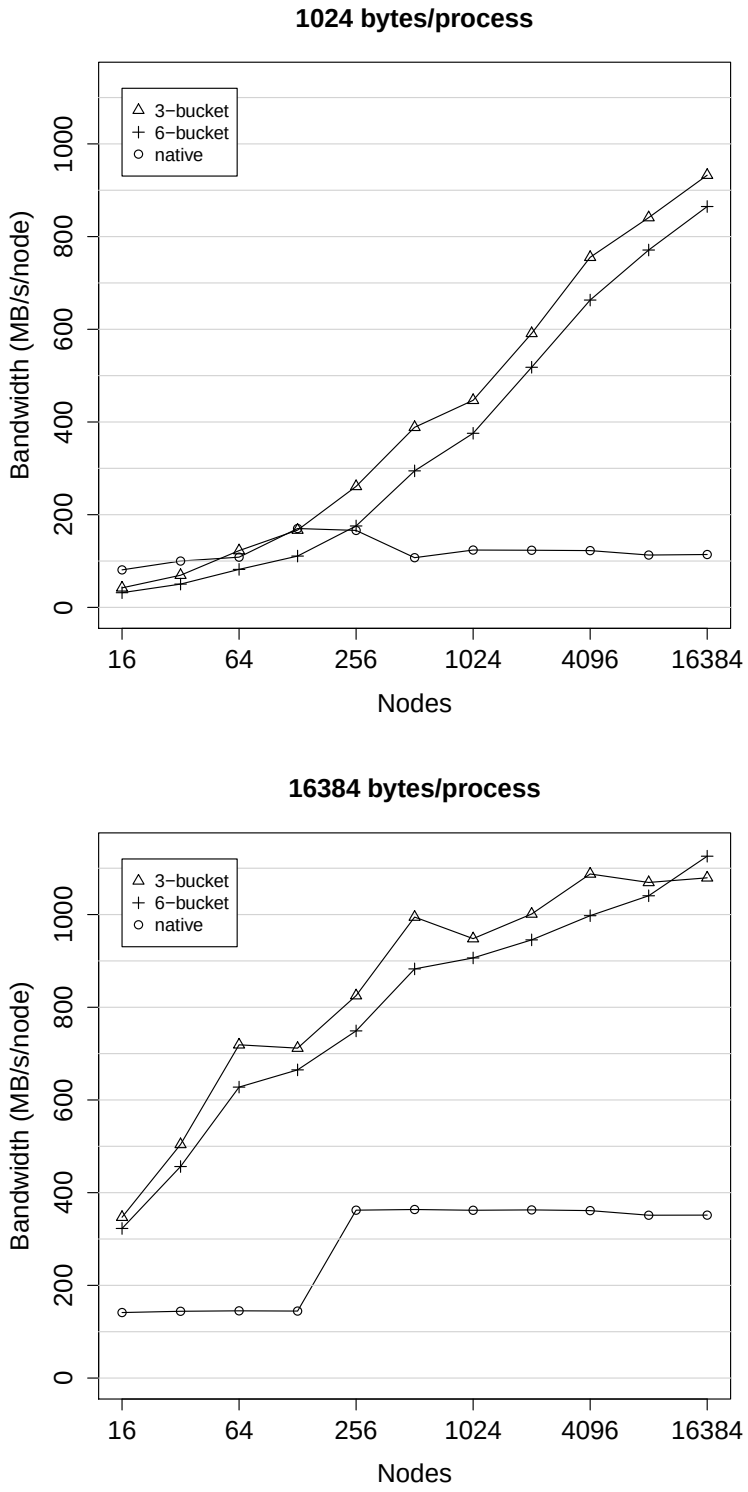

Figure 3.9: Performance of multi-port allgather algorithms *vs* number of processors on mesh subpartitions.

(upper graph) to be larger than the divergence for larger messages (lower graph).

Figure 3.9 also shows that that our parallel bucket algorithms deliver significantly better performance than the native algorithm with as few as 16 processors for sufficiently large input messages.

We also observed that the native algorithm performs worse for subpartitions than full partitions. This is because the faster tree network is not available on subpartitions, so the native allgather-via-broadcasts algorithm must use the torus network.

3.5.4 Reduce-scatter performance

Figures 3.10 and 3.11 show the performance of the reduce-scatter variations. As mentioned above, the communication pattern of reduce-scatter is the reverse of that of the allgather operation. The input vector on each process is arranged as 32-bit integer values and we sum them to form the output vector.

We again observed that the performance of the native algorithm on the default communicator, `MPI_COMM_WORLD` (native-world), was far better than on a copy (native-copy). We suspect the contiguous subpartition algorithm, which uses the torus network, is being used on copies of `MPI_COMM_WORLD` and on `MPI_COMM_WORLD` when the message is smaller than 512 bytes. For larger messages on `MPI_COMM_WORLD`, the library likely uses the tree network, which supports in-place reduction on integers.

The non-topology-aware recursive-halving, distance-halving algorithm (rh halving), performs the worst of the rest of the algorithms. The semi-topology-aware distance-doubling algorithm (rh doubling) performs better, followed by the optimally-remapped recursive-halving algorithm (rh optimal). The bucket and 6-bucket algorithms perform best, except for smaller messages with smaller allocations. The 6-bucket algorithm performs much better than the others, except for small messages with the smaller 512-node allocation.

Figure 3.12 shows the performance of the reduce-scatter algorithms on subpartitions. The native algorithm performs quite poorly for small and large messages alike compared to the rest of the algorithms.

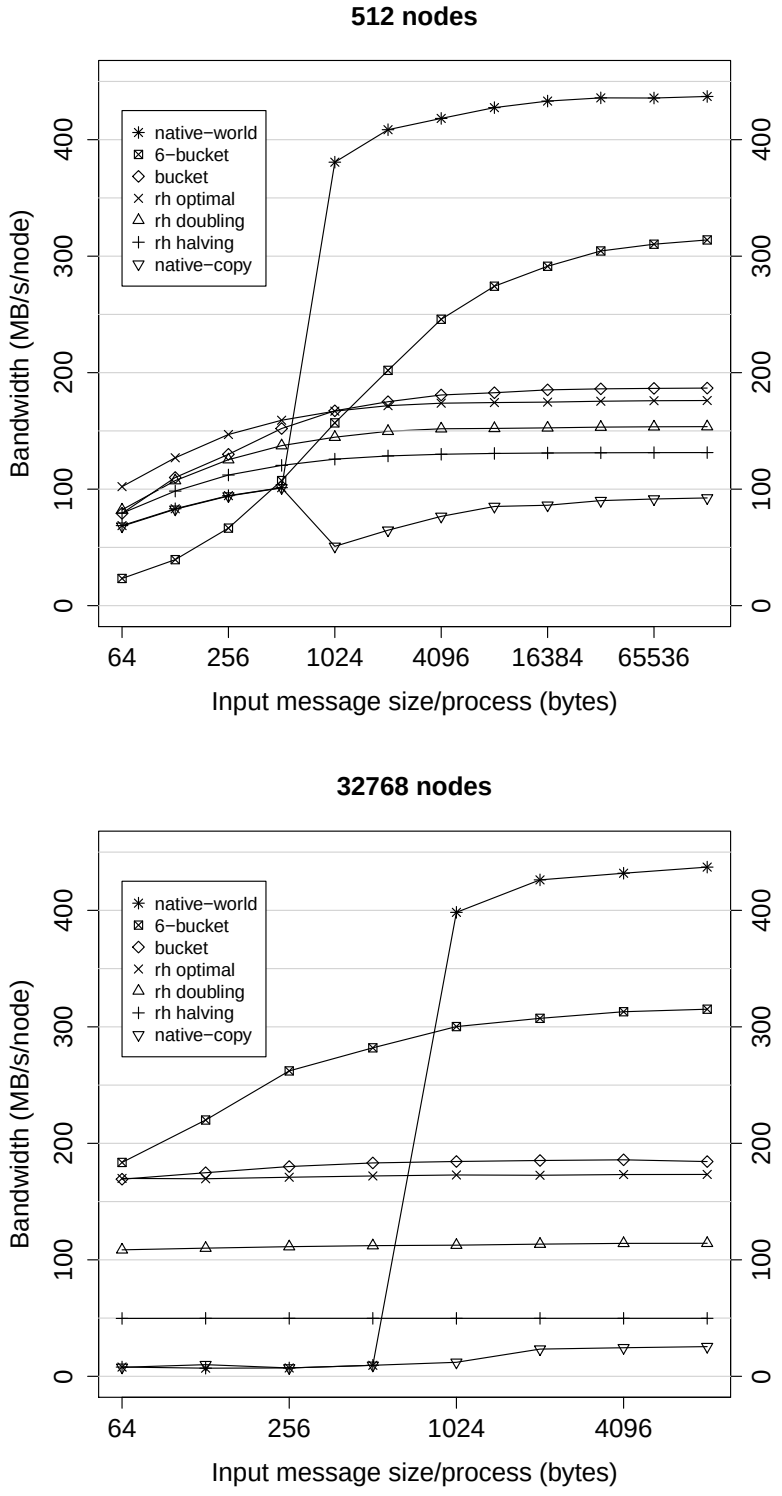

Figure 3.10: Performance of reduce-scatter algorithms *vs* message size.

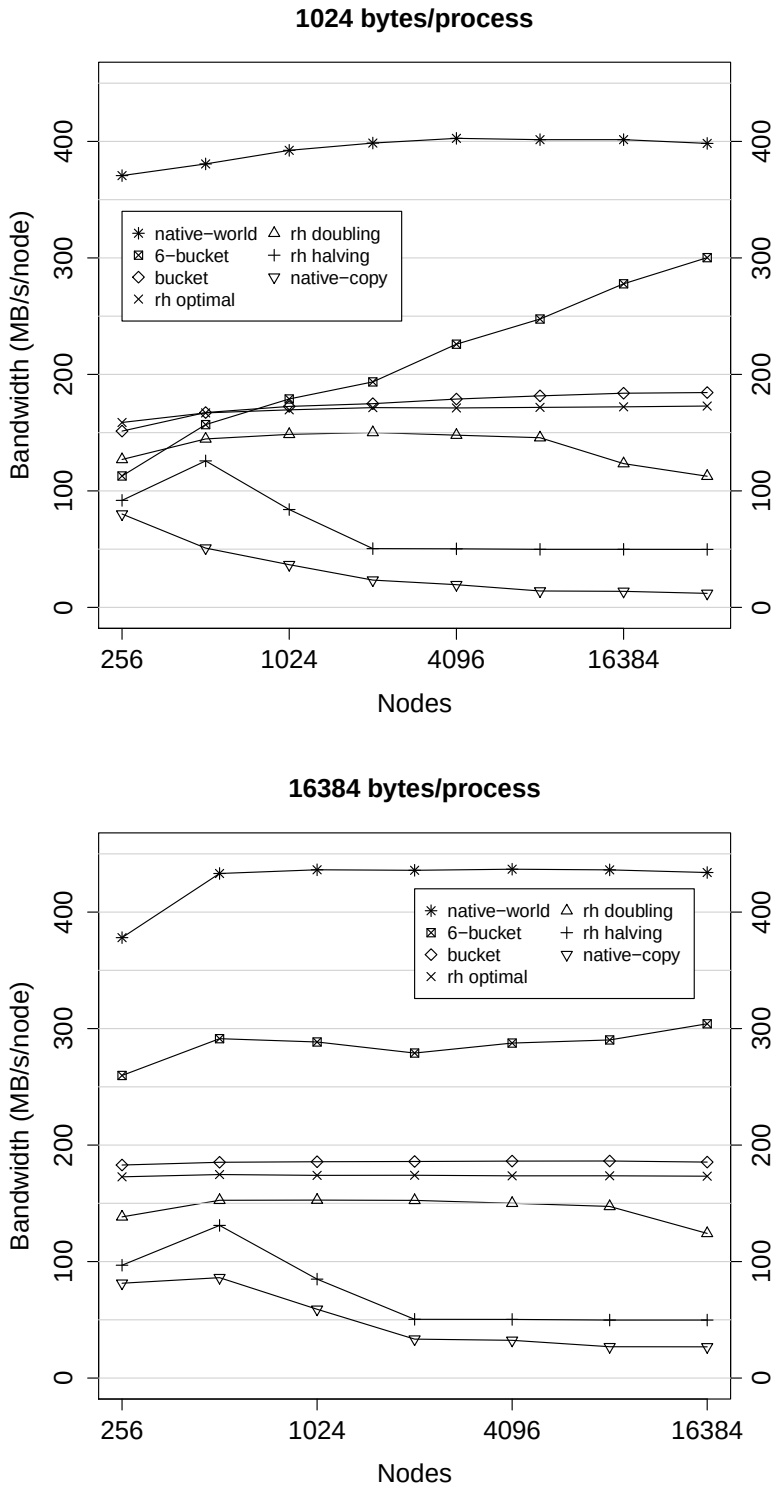

Figure 3.11: Performance of reduce-scatter algorithms *vs* number of processors.

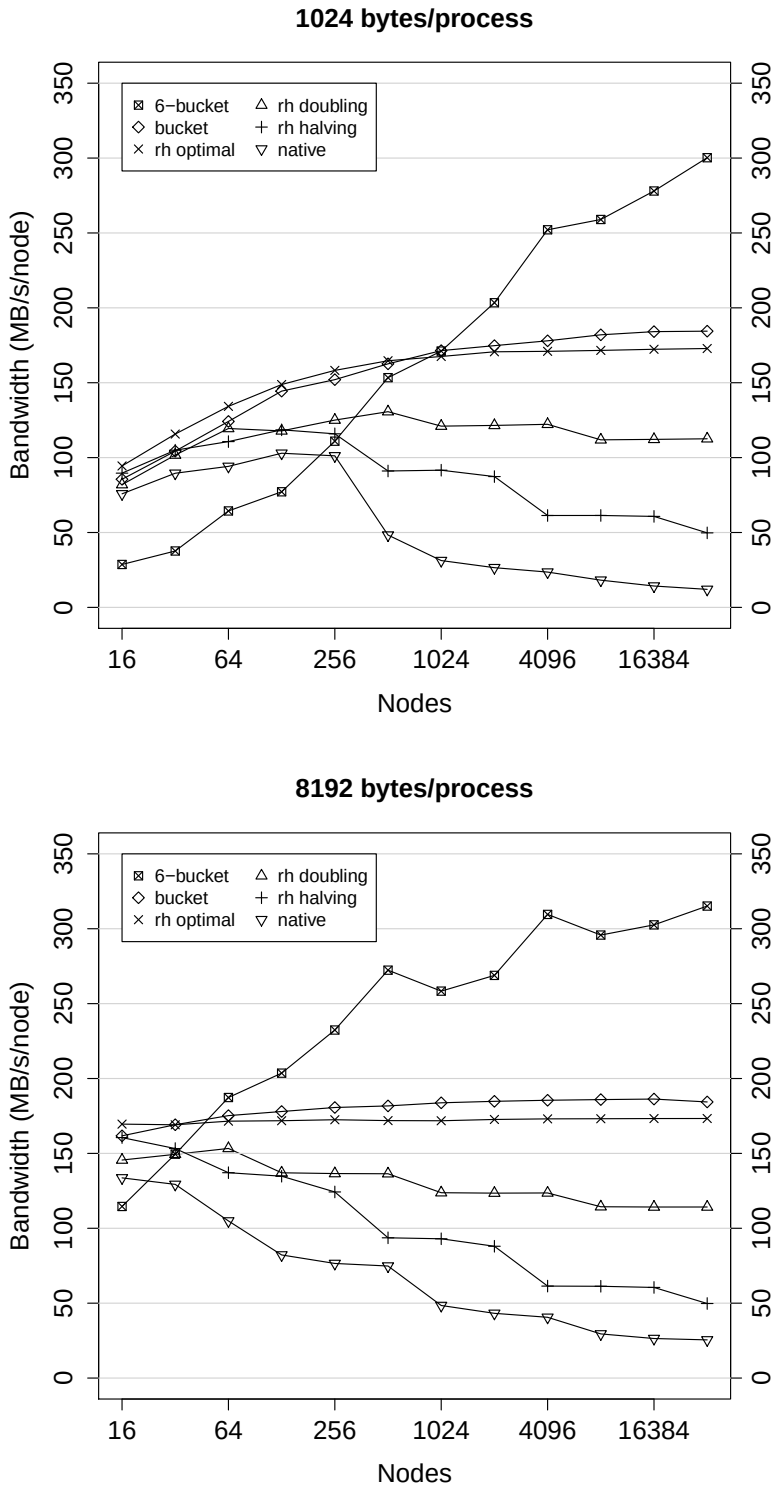

Figure 3.12: Performance of reduce-scatter algorithms *vs* number of processors on mesh subpartitions.

3.5.5 Broadcast performance

Figure 3.13 shows the performance of the broadcast algorithms. The bandwidth delivered by the native algorithm far exceeds the bandwidth delivered by the other algorithms we studied, delivering up to 940 MB/s. It uses the hardware broadcast feature to broadcast data to all nodes along a line in the torus network. The other algorithms are implemented as a native scatter followed by an allgather algorithm of our choosing.

As with the allgather operation alone, the 6-way bucket algorithm is generally best. The bucket algorithm is next-best, closely followed by the optimally-remapped recursive-doubling. They are followed by the semi-topology-aware distance-halving, recursive-doubling algorithm, delivering far more bandwidth than the non-topology-aware distance-doubling recursive-doubling algorithm.

3.5.6 Measured versus predicted performance

We now compare the measured performance for several allgather algorithms with the performance predicted by our model from Section 3.1. The δ term is simply the reciprocal of the link speed (375 MB/s) for one or three-port algorithms, and 7% less for six-port algorithms. The ring algorithm on the 32k-node partition consists of 32767 stages comprised of one message per stage per process and has a latency of 206 ms with a zero-byte input message. Thus, $\alpha = 206$ ms$/32767 = 6.29\mu$s.

In Section 3.4, we developed approximate equations for the performance of different allgather algorithms; we ignored minor terms, such as the extra messages our algorithms introduce. On large systems, these terms are negligible. In this analysis, we do not throw out any terms. Further, our equations assumed cubic torus networks. Here, we directly apply our congestion-aware performance model by counting messages and bytes; we do not use any equations. We will explain with two examples how we derive the predicted points in the figures to follow which evaluate the validity of our model.

Consider the performance of the optimally-reordered recursive-doubling algorithm on a 512-node 8x8x8 torus ($P = 512; X = Y = Z = 8$). The execution consists of nine stages of recursive-doubling after one stage of the input shuffle. Thus, the message startup cost is 10α. In the first three stages,

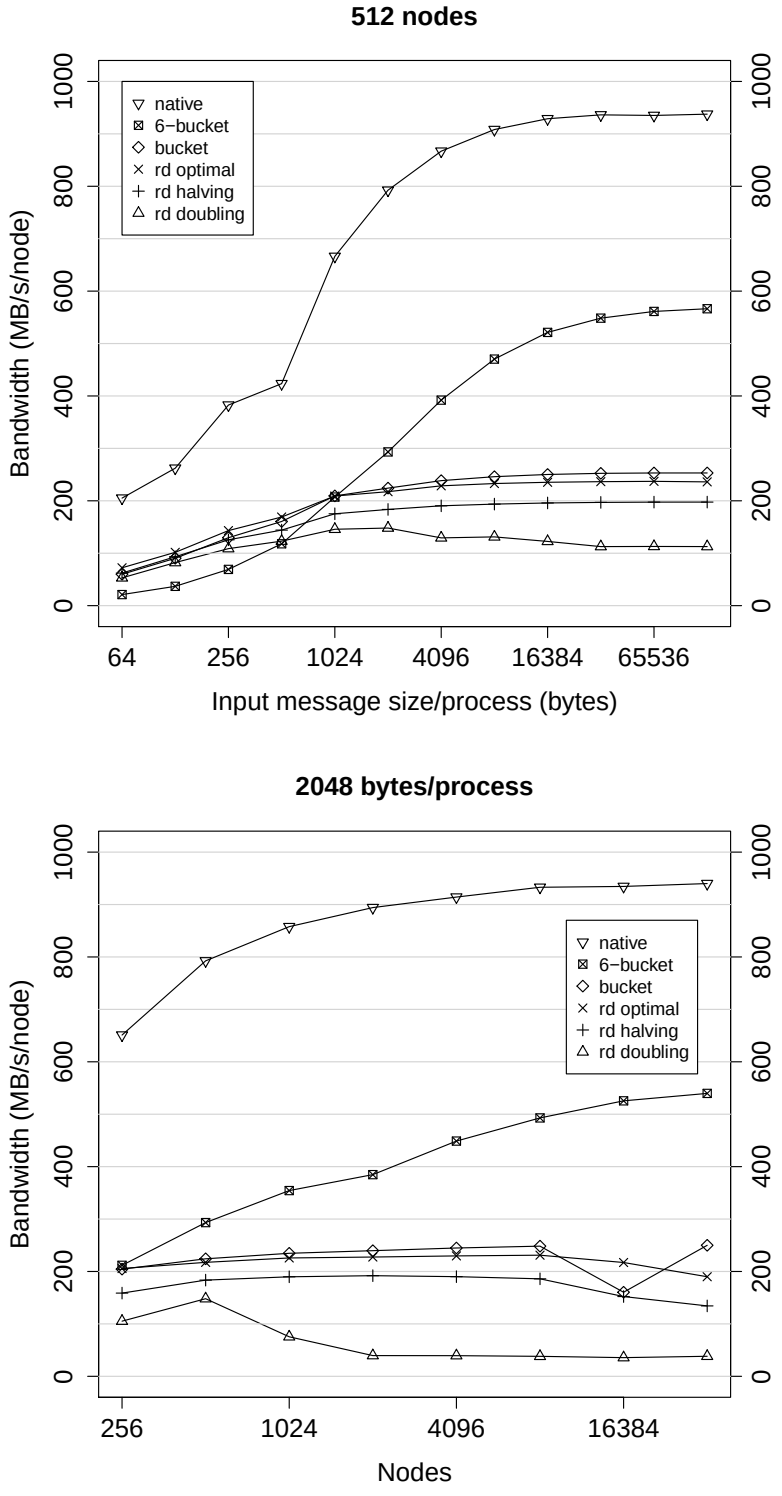

Figure 3.13: Performance of broadcast algorithms.

nodes four hops apart in each dimension exchange messages of size n, $2n$, and $4n$ bytes. They will incur a congestion penalty of two rather than four due to the use of wrap-around links. This is one detail ignored by our model for this algorithm, since on large torus systems, it is a negligible term. (For the recursive-doubling, distance-doubling algorithm, we account for the wrap-around links in our final equations.) In the middle three stages, nodes two hops apart in each dimension exchange messages of size $8n$, $16n$, and $32n$ bytes with a congestion penalty of two. In the last three stages, nodes one hop apart exchange messages of size $64n$, $128n$, and $256n$ with no congestion.

We do not have an analytical model for the input shuffle stage. Thus, we divide the total amount of data moved (nP) by the bidirectional bisection bandwidth ($4P^{2/3}$) and add $n\sqrt[3]{P}/4 = n\sqrt[3]{512}/4 = 2n$ to the bandwidth term. We could achieve tighter bounds on our estimation by modeling every message and every link for every partition size, but it would make little difference. The total cost is then:

$$\begin{aligned} C_{torus} &= (\log_2 512 + 1)\alpha + \\ &\quad n\delta\{2(1+2+4) + 2(8+16+32) + 1(64+128+256)\} + \\ &\quad 2n\delta \\ &= 10\alpha + 576n\delta. \end{aligned}$$

According to our prior equation, $C_{torus} = (\log_2 P + 1)\alpha + (7/6)nP\delta$, we would calculate $C_{torus} = 10\alpha + 597.33n\delta$.

Let us look at a more complex algorithm: the 6-way bucket algorithm on a non-cubic 4096-node 8x16x32 torus system ($P = 4096; X = 8; Y = 16; Z = 32$). In the first part of the algorithm, we round-up the message sizes to the next multiple of six bytes. This involves receiving up to $n + 5$ bytes in up to three messages or sending up to n bytes in two messages. Our model charges α once for each pair of send and receive messages; this exchange does not fit our model well. We will charge $3\alpha + n\delta$ for this step; the extra 5 bytes we will continue to ignore. In the second part, each process divides its input message into six portions and exchanges them with other processes. This involves an exchange of six messages accounting for n bytes.

For the first two phases, we do not have an analytical model for congestion.

Again, we divide the total amount of data transferred ($nXYZ$ bytes) by the minimal bisection bandwidth ($4XY$) for a bandwidth term of $nZ/4 = 8n$ for each phase.

Then, the main algorithm proceeds in three phases. In the first phase of the main algorithm, each process will exchange $2((X-1)+(Y-1)+(Z-1)) = 106$ messages. The bandwidth term will be bound by the ZXY buckets, which each circulate $(Z-1)n/6 = (31/6)n$ bytes.

In the second phase, which begins after all the processes complete the first phase, each process again exchanges $2((X-1)+(Y-1)+(Z-1)) = 106$ messages. The bandwidth term will be bound by the YZX buckets, which each circulate $(Z-1)Yn/6 = (446/6)n$ bytes.

In the final phase, 106 messages are exchanged per process and the bandwidth term is bound by the XYZ buckets, which each circulate $(Z-1)XYn/6 = (3968/6)n$ bytes.

The total cost is then:

$$\begin{aligned} T_{torus} &= (3+6+3\cdot 106)\alpha + n(8+8+(31/6)+(446/6)+(3968/6))\delta \\ &= 327\alpha + (4591/6)n\delta. \end{aligned}$$

This is a non-cubic partition and our prior equation for 6-bucket performance assumed cubic partitions and does not apply.

Figure 3.14 compares the performance predicted by our model for the ring algorithm versus the actual performance on Intrepid for small 64-byte messages and large 32-kilobyte messages. We computed α using this algorithm, hence, for the α-dominated 64-byte input message problem, the model agrees almost perfectly. For the 32-kilobyte input message problem, our model predicts better performance than we observe.

Figure 3.15 evaluates our model on the recursive-doubling, distance-doubling algorithm and recursive-doubling, optimally-reordered algorithm, and Figure 3.16 evaluates our model on the one-port and six-port bucket algorithms. Recall that for the 6-bucket case, we adjust δ to account for the 7% penalty for using all 6 ports at the same time. Thus, $\delta_{6bucket} = \delta/0.93$.

Overall, we see that the model agrees very well for large messages with the three topology-aware algorithms: the reordered recursive-doubling algorithm and the two bucket algorithms. For short messages on all algorithms and short and long messages on the ring algorithm, the startup cost becomes

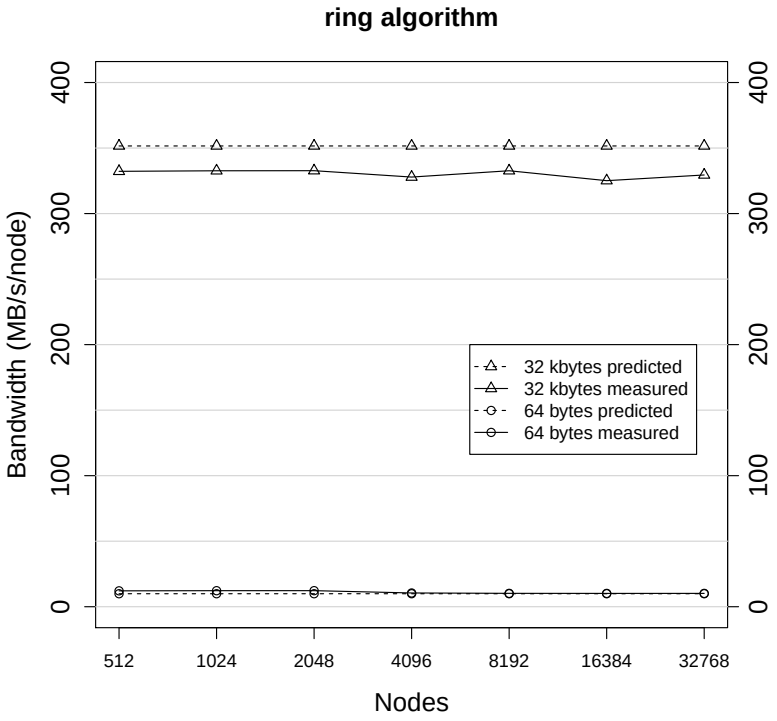

Figure 3.14: Predicted and measured performance of ring allgather algorithm.

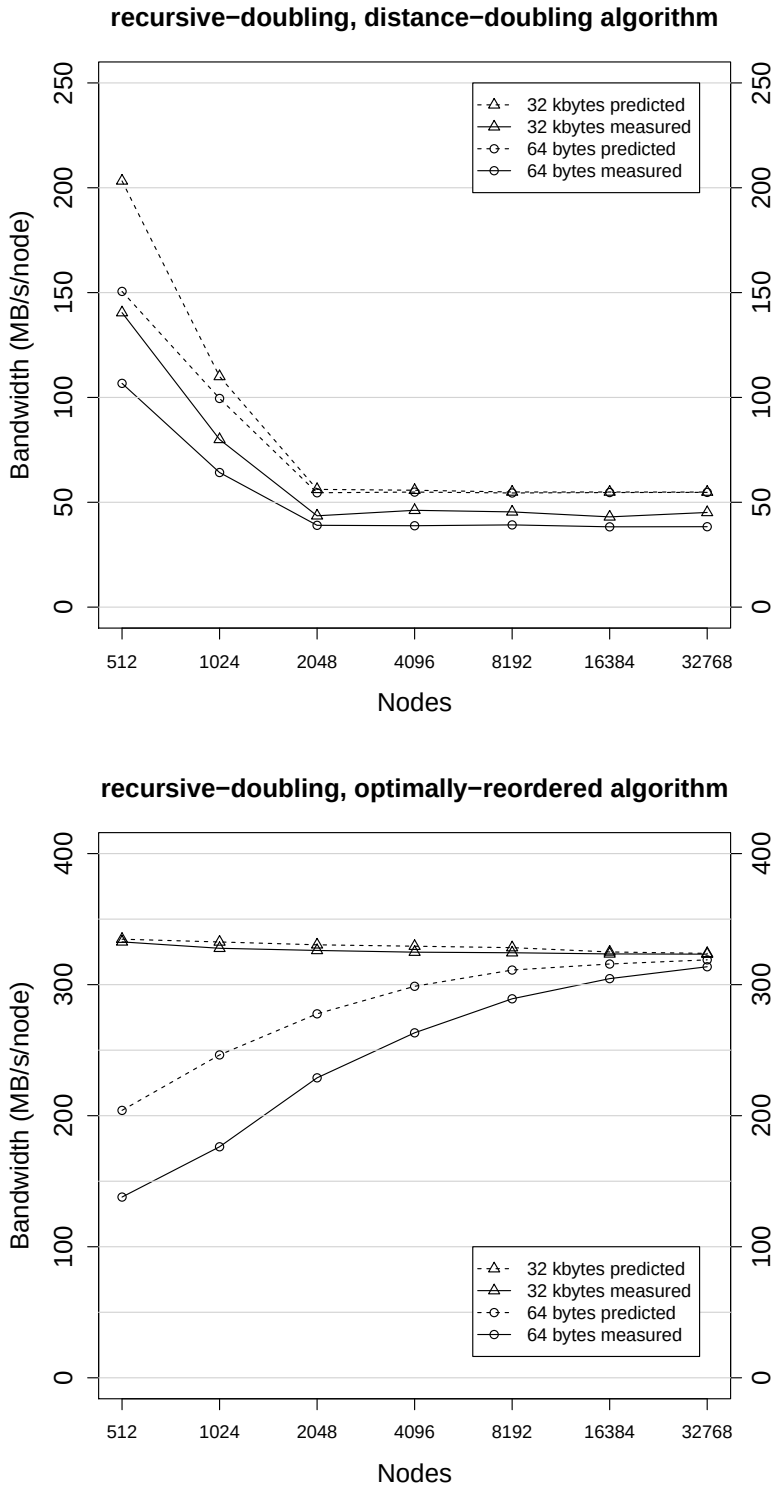

Figure 3.15: Predicted and measured performance of recursive-doubling, distance-doubling and recursive-doubling, optimally-reordered allgather algorithms.

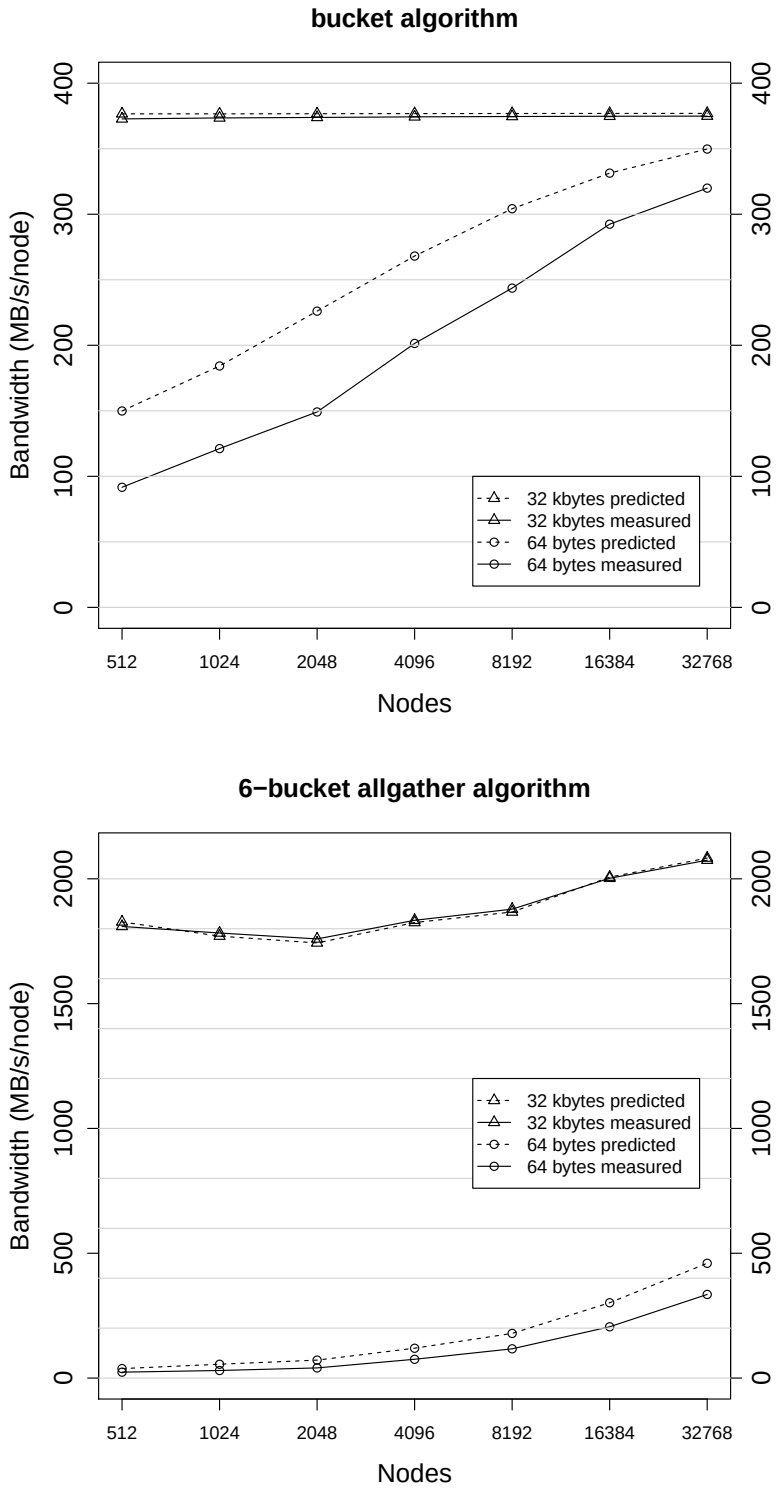

Figure 3.16: Predicted and measured performance of bucket and 6-way bucket allgather algorithms.

more important and our model loses some fidelity. Nonetheless, there is a clear pattern between the performance predicted by our model and the realized performance on Intrepid.

3.5.7 Comparison with similar work

As mentioned above in Section 3.4.2, Jain and Sabharwal also implemented 6-way bucket algorithms [31]. Their evaluation was also on a Blue Gene/P system. Their multi-threaded allgather algorithm delivers about the same performance as our single-threaded implementation with an extra stage. Their single-threaded algorithm performs about 30% worse. On the other hand, their multi-threaded reduce-scatter algorithm performs much better than ours, since they can spread the application of the reduction operator over multiple threads. Their single-threaded reduce-scatter algorithm has similar performance to ours. Instead of an extra stage of communication, they reorder the data in memory after the allgather algorithm and before the reduce-scatter algorithm, as we do in the experiment whose results are shown in Figure 3.8. It is unclear to us how they can do this reordering so efficiently, since we tried this and performance suffered greatly, as Figure 3.8 shows.

3.6 Conclusion

3.6.1 Other non-minimal algorithms

Hybrid algorithm: The authors of [30] found that the best allreduce performance on a 2-d mesh involved a hybrid of the bidirectional exchange and recursive-halving algorithms. The 6-bucket algorithm performs very well for larger problems but less well for small messages due to the large number of messages. We implemented a hybrid triple-recursive-doubling/sextuple-ring algorithm.

On a cubic torus, it consists of three simultaneous recursive-doubling operations. Each recursive-doubling operation operates on two of the three dimensions, and the dimensions are staggered so that the recursive-doubling

operations do not compete for links. This is followed by the last stage of the 6-bucket algorithm: a ring in the third dimension.

This would reduce the number of messages from approximately $18\sqrt[3]{P}$ to $2\lg P + 6\sqrt[3]{P}$.

We evaluated this algorithm on an 8x8x8 torus partition on Intrepid. In this case, the number of messages is reduced from 127 to 59, a factor of 2.15. Unfortunately, we found that the performance of the hybrid was only better than the 6-bucket algorithm for very small messages where the hybrid was slower than the better single-ported algorithms. On a 32x32x32 32k-node partition, the number of messages would be reduce from 559 to 217, a factor of 2.57. This algorithm might be useful on an exascale supercomputer with much larger dimensions.

3.6.2 Summary

Many widely-used allgather and reduce-scatter algorithms do not reach full potential communication performance for several reasons. They are single-ported or not topology-aware. In all of them, processes send and receive the minimal amount of data. In contrast, our algorithms are non-minimal because we add an extra stage of redundant communication to restore the correct data order. This flexibility allows us to reorder the stages of communication or run multiple operations in parallel on a multi-port network. The overhead of the extra stage is small and this leads to substantially better performance compared to minimal-communication algorithms.

In sum, our contribution is two-fold. We show that collective algorithms with redundant communication can deliver far better performance than known collective algorithms. We also show that collective algorithms need not preserve the correct ordering as long as the misordering is the same on all the processes, since a simple extra stage can restore the correct ordering. We present a novel semi-topology-aware recursive-doubling/distance-halving allgather algorithm and a recursive-halving/distance-doubling reduce-scatter algorithm that work well on a variety of networks and are optimal on Clos networks; a novel topology-aware reordered recursive-doubling allgather and reordered recursive-halving reduce-scatter algorithm for single-port mesh or torus networks; and a novel multi-port bucket algorithm for mesh or torus

networks. Our allgather algorithm delivers within 1% of the maximum deliverable multi-port bandwidth of a Blue Gene/P system, which is 5.5x better than the native algorithm. Our reduce-scatter algorithm delivers up to 11x better performance than the native algorithm when the native algorithm is not using the tree network.

Chapter 4

Conclusion

This thesis makes original contributions in two areas: process partitioning and collective algorithms.

Process partitioning and remapping: We demonstrate that existing `MPI_Comm_split` algorithms simply will not scale to exascale systems, either in time or space. We analyze a novel `MPI_Comm_split` algorithm that does, by replacing the sorting at the end with merging after each step, sorting in parallel, and distributing the final table. This results in a projected speedup of 60x on the largest exascale system we consider and an arbitrary benefit in space.

Collective algorithms: We show that known algorithms for several important collective-communication operations do not perform well when topology is considered. Algorithms for Clos networks are ordered so that the worst congestion occurs when the largest messages are exchanged. Algorithms for torus or mesh networks only use one port when the use of multiple ports can deliver much higher performance.

Our algorithms differ from the known in that they involve redundant communication. This gives us the flexibility to reorder the stages of communication in the case of the recursive-doubling or recursive-halving algorithms, or to run multi-port algorithms on multi-ported torus networks. We provide a generalized technique that rearranges the communication stages in these algorithms that reduces network load and congestion, improving performance.

On a 32k-node system, this gives us a speedup over the native algorithm of up to 5.5x for the allgather operation and up to 11x for the reduce-scatter operation. Moreover, our algorithms do not *only* address the performance of collective operations on large partitions of large systems. Our results hold for 512-node torus networks and 16-node mesh networks as well. This is because the redundant communication, at worst, consumes 7% of the improved runtime of each operation with 512 nodes and 2.6% with 32k-nodes. This is

a small penalty compared to the benefits of reducing congestion or using all the ports available in a multi-ported network.

An important aspect of both portions of this thesis is that our novel algorithms do not require special hardware, special compilers, special runtime-support, or tuning. We contribute straight-forward algorithms that dramatically improve the performance of communicator-management and several important collective operations without any strings attached.

References

[1] D. S. Wise, "Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free," in *In Euro-Par 2000 Parallel Processing*. Springer, 2000, pp. 774–784.

[2] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, *TOP500 Supercomputing Sites*, 2011 (accessed September 13, 2011), `http://top500.org`.

[3] A. Toor, *Fujitsu K supercomputer now ranked fastest in the world, dethrones China's Tianhe-1A*, 2011 (accessed October 13, 2011), Engadget, `http://www.engadget.com/2011/06/20/fujitsu-k-supercomputer-now-ranked-fastest-in-the-world-dethron/`.

[4] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 41:1–41:11.

[5] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," in *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 165–172.

[6] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The PERCS high-performance interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 75–82.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[8] W. D. Gropp and E. Lusk, *User's Guide for* `mpich`*, a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996, aNL-96/6.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[10] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257-267 10th European PVM/MPI User's Group Meeting.* Springer Verlag, 2003, pp. 257–267.

[11] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[12] D. R. Cheng, A. Edelman, J. R. Gilbert, and V. Shah, "A novel parallel sorting algorithm for contemporary architectures," *technical report, University of California at Berkeley*, 2006.

[13] E. L. G. Saukas and S. W. Song, "A note on parallel selection on coarse grained multicomputers," *Algorithmica*, vol. 24, pp. 371–380, 1999.

[14] P. Sack and W. Gropp, "A scalable mpi_comm_split algorithm for exascale computing," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–10.

[15] A. Moody, D. Ahn, and B. de Supinski, "Exascale algorithms for generalized MPI_comm_split," in *Proceedings of the 18th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'11, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds., vol. 6960. Springer Berlin / Heidelberg, 2011, pp. 9–18.

[16] C. Siebert and F. Wolf, "Parallel sorting with minimal data," in *Proceedings of the 18th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'11, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, vol. 6960, pp. 170–177.

[17] H. Shi, R. Council, G. N. Ogp, J. Schaeffer, and J. Schaeffer, "Parallel sorting by regular sampling," 1992.

[18] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[19] N. Choudhury, Y. Mehta, T. L. Wilmarth, E. J. Bohm, and L. V. Kalé, "Scaling an optimistic parallel simulation of large-scale interconnection networks," in *WSC '05: Proceedings of the 37th conference on Winter simulation*. Winter Simulation Conference, 2005, pp. 591–600.

[20] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI collective communication on BlueGene/L systems," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 253–262.

[21] A. Faraj, S. Kumar, B. Smith, A. Mamidala, J. Gunnels, and P. Heidelberger, "MPI collective communications on the Blue Gene/P supercomputer: algorithms and optimizations," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 489–490.

[22] A. Bhatele, "Automating topology aware mapping for supercomputers," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.

[23] C. Leiserson and B. Maggs, "Communication-efficient parallel algorithms for distributed random-access machines," *Algorithmica*, vol. 3, pp. 53–77, 1988.

[24] R. Wankar and R. Akerkar, "Reconfigurable architectures and algorithms: A research survey." *International Journal of Computer Science and Applications*, pp. 108–123, 2009.

[25] H. Tang and T. Yang, "Optimizing threaded MPI execution on SMP clusters," in *Proc. of 15th ACM international conference on supercomputing*. ACM Press, 2001, pp. 381–392.

[26] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[27] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[28] E. Zahavi, "Fat-trees routing and node ordering providing contention free traffic for mpi global collectives," *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, vol. 0, pp. 761–770, 2011.

[29] M. Barnett, D. G. Payne, and R. A. van de Geijn, "Optimal broadcasting in mesh-connected architectures," Austin, TX, USA, Tech. Rep., 1991.

[30] B. L. Payne, M. Barnett, R. Littlefield, D. G. Payne, and R. A. van de Geijn, "Global combine on mesh architectures with wormhole routing," in *Proc. of 7 th Int. Parallel Proc. Symp*, 1993.

[31] N. Jain and Y. Sabharwal, "Optimal bucket algorithms for large MPI collectives on torus interconnects," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 27–36.

[32] G. Bilardi, B. Codenotti, G. D. Corso, C. Pinotti, and G. Resta, "Broadcast and associative operations on fat-trees," 1996.

[33] P. Patarasuk and X. Yuan, "Bandwidth efficient allreduce operation on tree topologies," in *IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007.

[34] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: MPI's collective communication operations for clustered wide area systems," in *ACM SIGPLAN Notices*, 1999, pp. 131–140.

[35] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts, "Interprocessor collective communication library (intercom)," in *In Proceedings of the Scalable High Performance Computing Conference*. IEEE Computer Society Press, 1994, pp. 357–364.

[36] B. Juurlink, P. Kolman, and I. Rieping, "Optimal broadcast on parallel locality models," in *Proc. 7th Int. Coll. Structural Information and Communication Complexity, SIROCCO*, 2000.

[37] R. M. Karp, A. Sahay, E. E. Santos, K. E. Schauser, and A. S. E. E. Santos, "Optimal broadcast and summation in the LogP model," in *In Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, 1993, pp. 142–153.

[38] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12.

[39] V. Bala, J. Bruck, S. Member, R. Cypher, P. Elustondo, A. Ho, C. tien Ho, S. Kipnis, M. Snir, and S. Member, "CCL: A portable and tunable collective communication library for scalable parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, p. 164, 1995.

[40] R. Thakur and R. Rabenseifner, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[41] R. Rabenseifner, "A new optimized MPI reduce algorithm," `http://www.hlrs.de/mpi/myreduce.html`, 1997.

[42] J. L. Traff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Gropp, "A pipelined algorithm for large, irregular all-gather problems," *Int. J. High Perform. Comput. Appl.*, vol. 24, pp. 58–68, February 2010.

[43] T. Hoefler and J. L. Traeff, "Sparse Collective Operations for MPI," in *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, May 2009.

[44] J.-H. Lee, C.-S. Shin, and K.-Y. Chwa, "Optimal embedding of multiple directed hamiltonian rings into d-dimensional meshes," *Journal of Parallel and Distributed Computing*, vol. 60, no. 6, pp. 775 – 783, 2000.

[45] Richard and Stong, "Hamilton decompositions of cartesian products of graphs," *Discrete Mathematics*, vol. 90, no. 2, pp. 169 – 190, 1991.

[46] "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, pp. 199–220, January 2008.

[47] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[48] *IBM System Blue Gene Solution: Blue Gene/P Application Development*, 2008.

[49] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema, "Characterizing the performance of "big memory" on Blue Gene Linux," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ser. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 65–72.